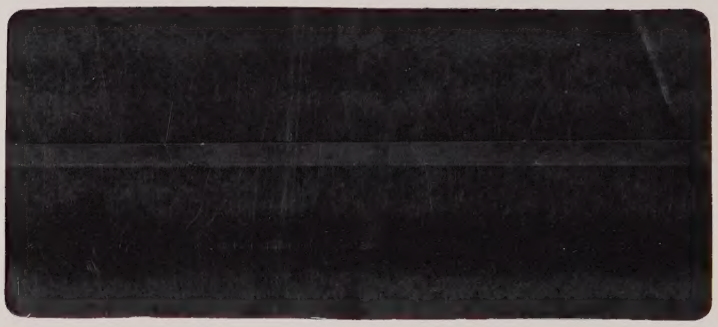
KAY



GPIB-488 Functions, model 5519

for interfacing with a DSP Sona-Graph™, model 5500

February 1990

Kay Elemetrics Corp
12 Maple Avenue
Pine Brook, NJ 07058-9798

Tel: 201/227-2000 • FAX: 201/227-7760 • TWX: 710/734-4347

Contents

1 KEY_DOWN 1

2 ALL_KEYS_UP 3

3 LOCK_5500_KBD 5

4 UNLOCK_5500_KBD 6

5 MEMORY_CLEAR 7

6 CURSOR_UPDATE_DISABLE 8

7 CURSOR_UPDATE_ENABLE 9

8 EXPRESS_START 10

9 EXPRESS_STOP 11

10 RECALL_SETUP_TABLE 12

11 WINDOW_ON 13

12 WINDOW_OFF 15

13 GET_5500_VARBLOCK 17

14 GET_ANFORM 19

15 GET_SCREEN_CHANNEL 22

16 BUFFER_SIZE 24

17 GET_CURS_VALS 26

18 GET_TIME_SLICE 28

19 GET_VCBFLAG 30

20 GET_BUFFER 32

21 SEND_BUFFER 35

22 SCSI_ERROR 38

23 GET_SAMP_FREQ 40

24 CHANGE_IEEE_ADDRESS 42

25 INIT_TRANS 43

A DEFINES FOR ANALYSIS FORMAT TYPES 44

Digitized by the Internet Archive
in 2025

https://archive.org/details/manualsplus_10526

B DEFINES FOR OTHER CONSTANTS 44

C DEFINES FOR 5500 KEY CODES 45

D DEFINES OF STRUCTURES USED BY THE TRANSLATOR 47

E CURSOR DATA INFORMATION 49

F SCSI ERROR CODES 51

G Kay File Format 52

How to use the 5519 GPIB Translator

The 5519 GPIB Translator is a high-level interface to the 5500. By sending small blocks of properly arranged data ("Command Blocks"), to the Translator from any IEEE controller, that controller can access a library of function calls that perform a variety of useful tasks on the 5500. The Translator can be a TALKER or LISTENER, uses primary IEEE-488 addressing only, and is NEVER a CONTROLLER, only a PERIPHERAL. The hardware requirements are as follows:

- a Kay Elemetrics DSP Sona-graph model 5500 set up with a computer interface option.
- the 5519 GPIB Translator.
- a programmable IEEE-488 controller.

This manual contains the reference material needed to assemble command block data, how to "handshake" with the Translator via the SRQ line and SERIAL POLLS, and how to interpret what the Translator sends back. The manual also includes source code for a testing program written for diagnostic use at Kay Elemetrics. The program was written in Microsoft 'C' on an IBM-PC/XT/AT compatible, and utilizes high level "driver" calls provided with the IO-TECH Personal-488 IEEE Controller Card. IO-TECH manufactures IEEE-488 hardware for a variety of computer platforms and provides a fairly consistant driver interface across that line. If you do not use IO-TECH's equipment but another high-level product instead, the test source code could still be useful with substitution of equivalent driver calls.

Notice that the term *integer* that appears in this manual refers to a *16 BIT* signed integer.

Technical Information Concerning Transfer Of Data On The IEEE488 Bus

A — **Addressing the Translator:** The software in the translator recognizes device number 22 as its default IEEE488 address. To change the address, issue the "CHANGE_TRANS_ADDRESS" call described in the call library section.

B — **Service Requests:** This is a powerful IEEE488 feature that is heavily utilized by the translator software. After every READ or WRITE to or from the translator, the SRQ line will be asserted. The active controller then SERIAL POLLS the translator, making sure it was the source of the SERVICE REQUEST. If it was, the translator will make available a byte of information which represents the program status in the translator. This system provides a check to see if data is succesfully being written to and read by the translator, if the translator expects to send or receive more data, or if the translator is finished transferring data. Refer to "SERVICE REQUEST CODE LISTING" for their definitions.

C — **Errors:** Since the translator is making use of a C-Language library of SCSI/5500 routines, there are actually several places errors can arise. If the SRQ byte from the translator indicates a SCSI error, text describing the error can be sent over the IEEE bus for a deeper investigation. There may, for instance, be an unconnected cable from the translator to the 5500. The error text "Target selection timeout" would indicate that problem. Other errors might be incorrect or illegal command lines, or a logic error causing the translator to be in one mode (i.e. LISTENING), while the active controller wants the translator to SEND data. The translator will fall out of the routine it is in when this sort of conflict occurs, and issue a SRQ byte indicating a "MODE ERROR."

D — **Command Lines:** The command line is the initial block of several bytes that informs the translator which call you are going to attempt to make. In several cases, (i.e. KEY_DOWN), the command line is all that is needed to make the call. The remainder of the calls require additional data to or from the translator to make them complete. All the command line specifications are detailed in the call library section, as well as what the translator expects to happen after the command line is sent.

E — **Commmand Line Parameter Count:** This number, specified in each call description, represents the total number of parameters in the command line, including the command line I.D. code and itself. It is a 16-bit integer value.

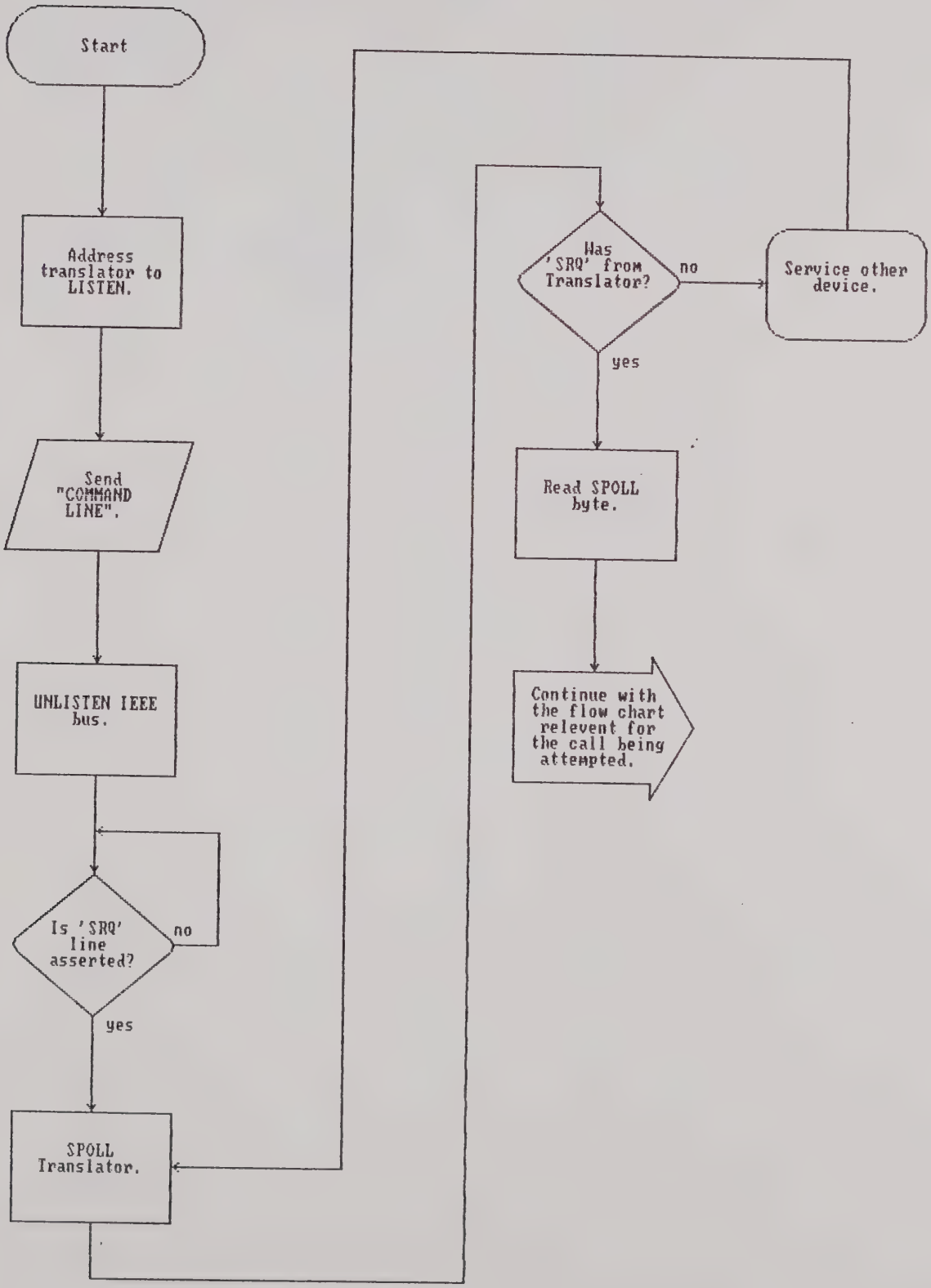
F — **Number of Bytes to Send:** Because of the various lengths in the number of bytes that are transferred to and from the translator, it uses the EOI, (END OR IDENTIFY), line to signify the last byte being sent or received. The largest block it will attempt to read or write in a large multibyte transfer is FFF0 hex bytes, or 65520 decimal bytes. (64K - 16 bytes). The active controller program should have a 64K buffer available for this data.

G — **The First Thing to Do:** Is send the INIT_TRANS call to the translator, before using any other calls. This will assure proper byte alignment for the other calls.

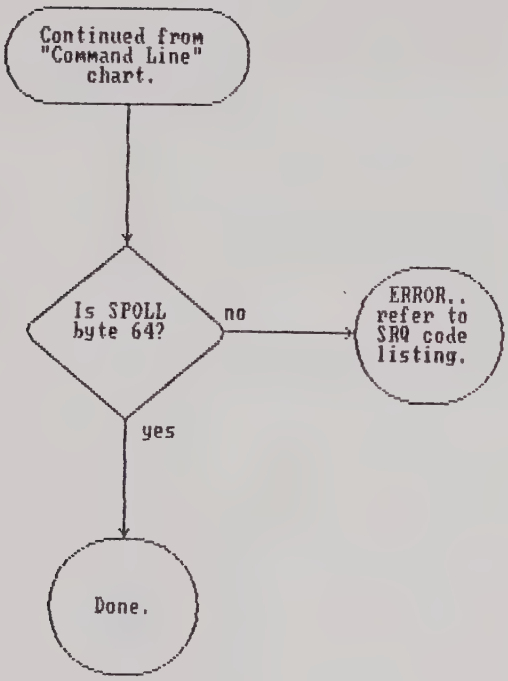
Reference To Flow Chart Diagrams

	CALL NAME:	FLOW CHART TO USE:
1 .	KEY_DOWN	CHART 2 (NO ADDITIONAL DATA REQUIRED)
2 .	ALL_KEYS_UP	CHART 2 (NO ADDITIONAL DATA REQUIRED)
3 .	LOCK_5500_KBD	CHART 2 (NO ADDITIONAL DATA REQUIRED)
4 .	UNLOCK_5500_KBD	CHART 2 (NO ADDITIONAL DATA REQUIRED)
5 .	MEMORY_CLEAR	CHART 2 (NO ADDITIONAL DATA REQUIRED)
6 .	CURSOR_UPDATE_DISABLE	CHART 2 (NO ADDITIONAL DATA REQUIRED)
7 .	CURSOR_UPDATE_ENABLE	CHART 2 (NO ADDITIONAL DATA REQUIRED)
8 .	EXPRESS_START	CHART 2 (NO ADDITIONAL DATA REQUIRED)
9 .	EXPRESS_STOP	CHART 2 (NO ADDITIONAL DATA REQUIRED)
10 .	RECALL_SETUP_TABLE	CHART 2 (NO ADDITIONAL DATA REQUIRED)
11 .	WINDOW_ON	CHART 4 (SMALL DATA TRANSFERS TO THE 5500)
12 .	WINDOW_OFF	CHART 4 (SMALL DATA TRANSFERS TO THE 5500)
13 .	GET_5500_VARBLOCK	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
14 .	GET_ANFORM	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
15 .	GET_SCREEN_CHANNEL	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
16 .	BUFFER_SIZE	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
17 .	GET_CURS_VALS	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
18 .	GET_TIME_SLICE	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
19 .	GET_VCBFLAG	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
20 .	GET_BUFFER	CHART 5 (LARGE DATA TRANSFERS FROM THE 5500)
21 .	SEND_BUFFER	CHART 6 (LARGE DATA TRANSFERS TO THE 5500)
22 .	GET_SCSI_ERROR	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
23 .	GET_SAMP_FREQ	CHART 3 (SMALL DATA TRANSFERS FROM THE 5500)
24 .	CHANGE_TRANS_ADDRESS	CHART 2 (NO ADDITIONAL DATA REQUIRED)

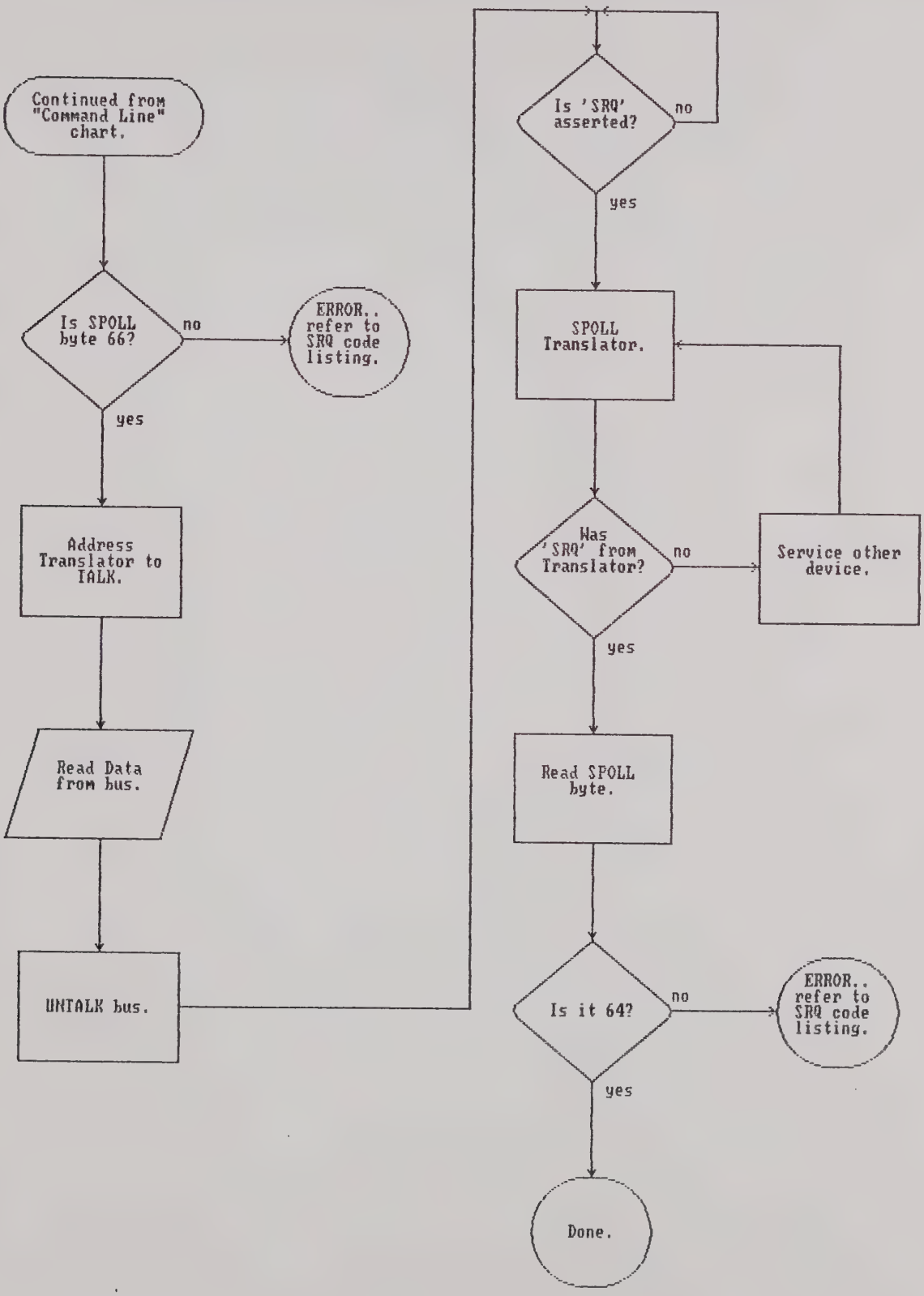
FLOW CHART #1: "COMMAND LINE" PROTOCOL WHICH BEGINS ALL CALLS.



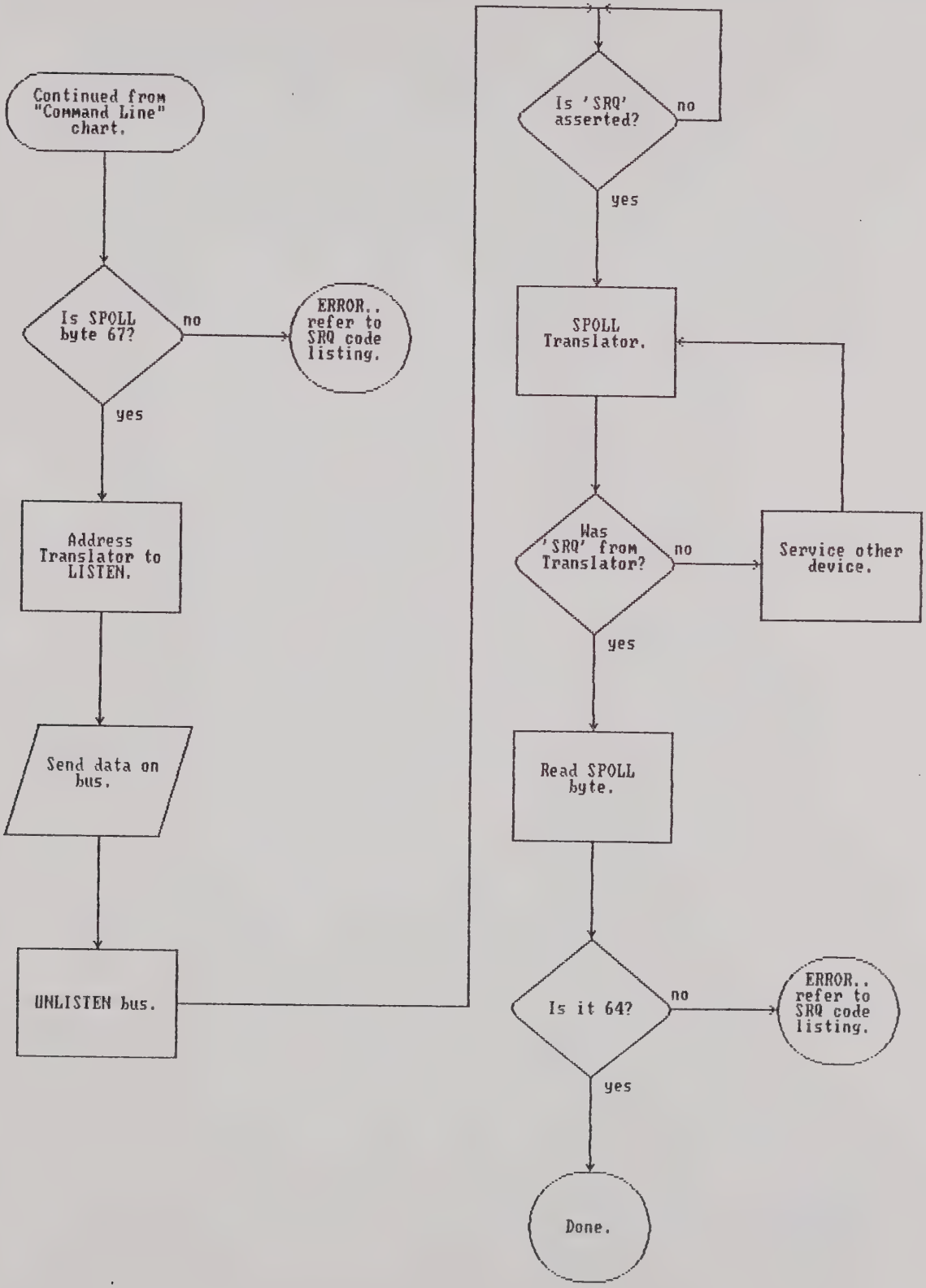
FLOW CHART #2: CALLS WHERE NO ADDITIONAL DATA IS REQUIRED.



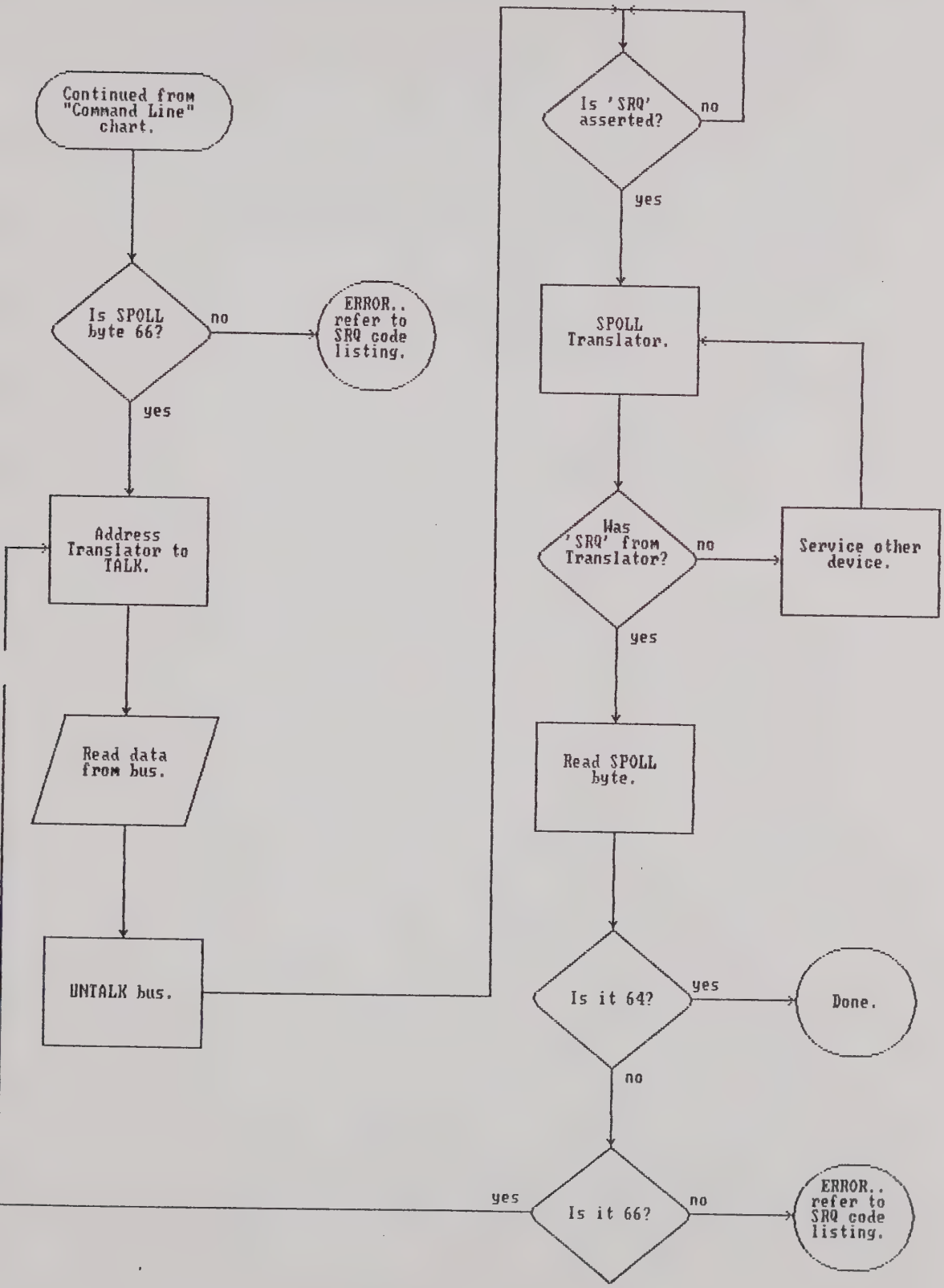
FLOW CHART #3: CALLS WITH SMALL (SINGLE) DATA TRANSFERS FROM THE 5500.



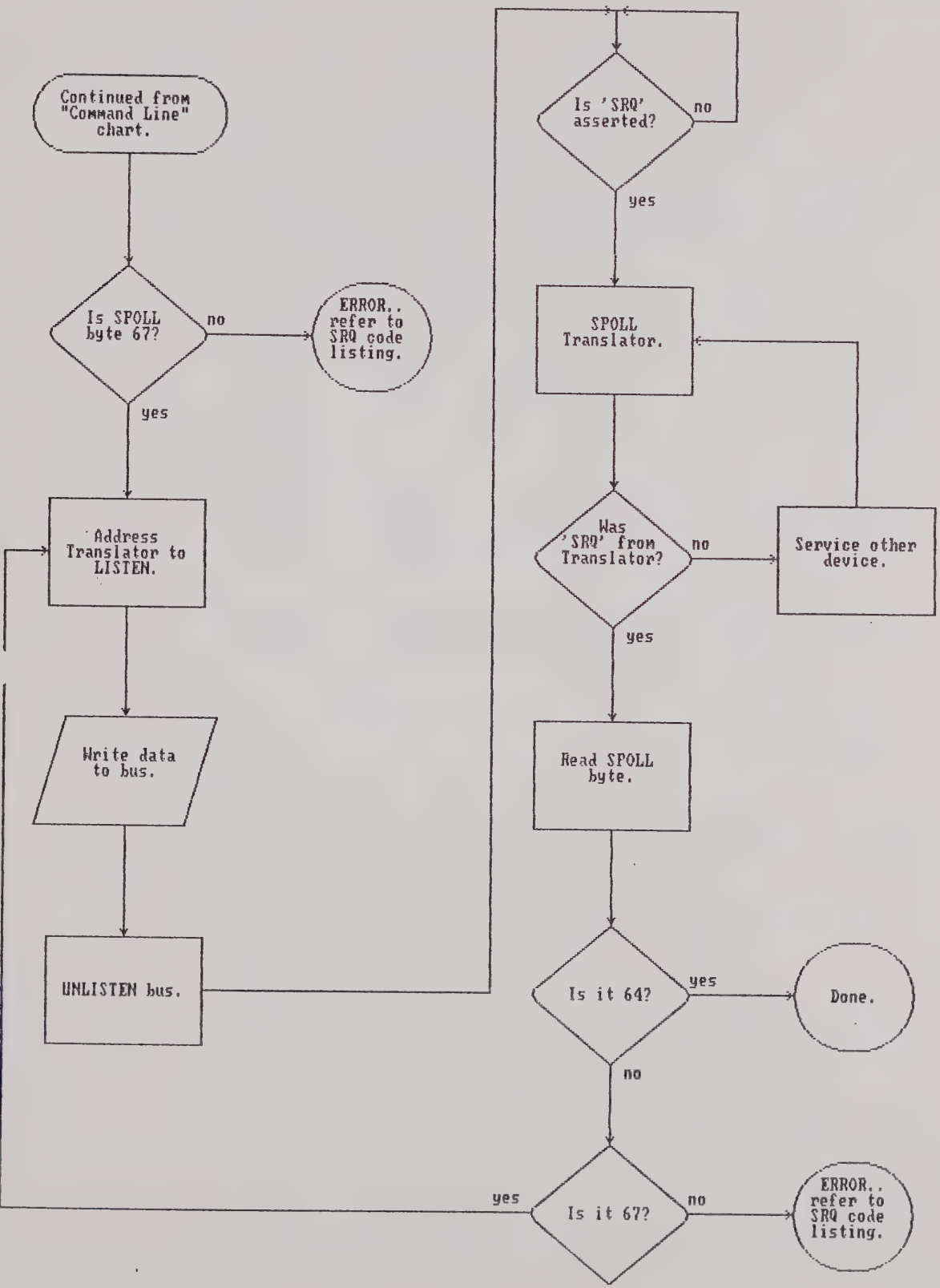
FLOW CHART #4: CALLS WITH SMALL (SINGLE) DATA TRANSFERS TO THE 5500.



FLOW CHART #5: CALLS WITH LARGE (MULTIPLE) DATA TRANSFERS TO THE 5500.



FLOW CHART #6: CALLS WITH LARGE (MULTIPLE) DATA TRANSFERS FROM THE 5500.



Summary of Available 5500 Calls

1. **KEY_DOWN:** The KEY_DOWN call sends codes to the 5500 to simulate pressing front panel buttons. Most front panel functions can be controlled from an IEEE488 active controller using this function. ALL_KEYS_UP should be called immediately after KEY_DOWN.

2. **ALL_KEYS_UP:** Signals the 5500 that no keys are pressed. Calling this function ensures that the 5500 knows the key code previously sent has been released.

3. **LOCK_5500_KBD:** Tells the 5500 not to read its front panel or remote keyboard. The only function still available on the front panel is the hardware reset.

4. **UNLOCK_5500_KBD:** Tells the 5500 to read its front panel and remote keyboard again.

5. **MEMORY_CLEAR:** Clears the buffers of both channels in the 5500. This function is equivalent to simultaneously pressing the "RECORD ON" and "RECORD OFF" keys on the 5500 front panel. It is recommended that you clear the 5500 memory before transferring data to the 5500, assuming that you don't want the current contents of the 5500's buffers.

6. **CURSOR_UPDATE_DISABLE:** For some 5500 display modes the power spectrum is updated when the cursors are moved. This function disables that feature, which is particularly useful when moving a cursor multiple times.

7. **CURSOR_UPDATE_ENABLE:** Re-enables power spectrum updates when cursors are moved.

8. **EXPRESS_START:** Starts the specified screen(s) on the 5500 scrolling.

9. **EXPRESS_STOP:** Stops the specified screen(s) scrolling.

10. **RECALL_SETUP_TABLE:** Recalls the specified stored setup on the 5500. This is equivalent to entering that number on the "ENTRY" keys on the 5500 front panel or remote keyboard.

11. **WINDOW_ON:** Displays a custom message on the lower half of the 5500 screen.

12. **WINDOW_OFF:** Removes the previously displayed window.

13. **GET_5500_VARBLOCK:** Returns the current values of variables in the structure of type *keyvars_5500*. This structure is defined in appendix D. It provides a "status check" of some 5500 settings.

14. **GET_ANFORM:** Returns the analysis format for the upper or lower screen of the 5500. The `#defines` for the returned value are in appendix A.

15. **GET_SCREEN_CHANNEL:** Returns which channel is being displayed on the upper or lower screen of the 5500.

16. **BUFFER_SIZE:** Returns the number of samples in channel one or channel two of the 5500.

17 . **GET_CURS_VALS:** Returns the current values of variables in the structure of type *CURSOR_VALUES*. This structure is defined in appendix D on page 47.

18 . **GET_TIME_SLICE:** Returns a spectrum from the 5500. Upper screen, Lower screen, and Red cursor are possible data sources.

19 . **GET_VCBFLAG:** Returns the location of the time cursors on the 5500. (Upper or lower screen).

20 . **GET_BUFFER:** Returns a data buffer from the 5500. The data is in binary-complement form, with a specified number of bits of resolution. This call is a LARGE (MULTIPLE) data transfer. The active controller will need to spool this data as it arrives (in blocks of FFF0 hex bytes maximum) to a file that can accomodate it.

21 . **SEND_BUFFER:** Sends a data buffer to the 5500. The data must be in binary-complement form, with number of bits of resolution specified. This call is a LARGE (MULTIPLE) data transfer. The translator will not accept data transfers of over FFF0 hex bytes at a time, so the active controller must make provisions for sending the data in these block sizes.

22 . **SCSI_ERROR:** Returns a string of text reporting the result of the most recent SCSI call made by the translator.

23 . **GET_SAMP_FREQ:** Returns the sampling frequency of a given buffer. (Channel ONE or Channel TWO).

24 . **CHANGE_TRANS_ADDRESS:** Sets the IEEE488 hardware in the translator to respond to a given address. The default address is "22".

25 . **INIT_TRANS:** Initializes the translator software. This call should be the first call made before any others.

Service Request Code Listing

SRQ NUMBER (decimal)	DEFINITION
#64	Command received OK, No further data required. SCSI call executed OK. Call complete.
#65	Command received OK, error in SCSI.
#66	Command received OK, data is ready to be sent.
#67	Command received OK, waiting for data.
#68	Command code number OK, parameter count incorrect.
#69	Command code incorrect or illegal.
#71	Mode error. Translator did not expect to be addressed to LISTEN or TALK at this time. Translator has reset IEEE488 hardware and is waiting for new COMMAND LINE.

1 KEY_DOWN

Summary

Command Line:

byte offset	data required	data type
command + 0	key_down I.D. (1 dec)	integer.
command + 2	parameter count (6 dec)	integer.
command + 4	key1	unsigned int.
command + 6	number of repetitions	unsigned int.
command + 8	key2	unsigned int.
command + 10	number of repetitions	unsigned int.

Description

The KEY_DOWN function sends key codes to the 5500 to simulate pressing front panel buttons. Most panel functions can be controlled via the IEEE488 bus using this function. ALL_KEYS_UP should be called after this function.

The available key codes are listed in appendix C.

Parameters

key1: The key code for keypress #1.

num1: The number of times key1 is pressed.

key2: The key code for keypress #2.

num2: The number of times key2 is pressed.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration to press RECORD ON button.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 1;          /* key down ID */
        buffer[1] = 6;          /* parm count */
        buffer[2] = 14;         /* code for RECORD ON */
        buffer[3] = 1;          /* 'press' once */
        buffer[4] = 0xFF;       /* no key 2 */
        buffer[5] = 0;          /* zero when there is
                                        no key2 */

        sendCommand(buffer); /* user program to transmit
```

```
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        buffer[0] = 2;  /* all keys up ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCalls were successful.\n");
}
```

2 ALL_KEYS_UP

Summary

Command Line:

byte offset	data required	data type
command + 0	all_keys_up I.D. (2 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The ALL_KEYS_UP function signals the 5500 that no keys are pressed. This function is called after KEY_DOWN. It ensures that the 5500 knows the key code previously sent has been released.

Parameters

None.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration to press RECORD ON button.

```
main()
{
        int buffer[6];
        int spollByte;

        buffer[0] = 1;  /* key down ID */
        buffer[1] = 6;  /* parm count */
        buffer[2] = 14; /* code for RECORD ON */
        buffer[3] = 1;  /* 'press' once */
        buffer[4] = 0xFF;     /* no key 2 */
        buffer[5] = 0;       /* zero when there is
                                    no key2 */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        buffer[0] = 2;  /* all keys up ID */
```

```
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCalls were successful.\n");
}
```

3 LOCK_5500_KBD

Summary

Command Line:

byte offset	data required	data type
command + 0	lock_5500_kbd I.D. (3 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The LOCK_5500_KBD function tells the 5500 not to read the front panel or remote keyboard. The only function available on the front panel is the hardware reset.

Parameters

None.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration to lock 5500 keyboard.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 3;  /* lock_5500_kbd ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

4 UNLOCK_5500_KBD

Summary

Command Line:

byte offset	data required	data type
command + 0	unlock_5500_kbd I.D. (4 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The UNLOCK_5500_KBD function tells the 5500 to read the front panel or remote keyboard again. It is the reverse of LOCK_5500_KEYBOARD.

Parameters

None.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration to unlock 5500 keyboard.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 4;  /* unlock_5500_kbd ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                          buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");
}
```

5 MEMORY_CLEAR

Summary

Command Line:

byte offset	data required	data type
command + 0	memory_clear I.D. (5 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The MEMORY_CLEAR function clears the buffers on both channels of the 5500. This function call is equivalent to simultaneously pressing the "RECORD_ON" and "RECORD_OFF" keys on the 5500 front panel. It is recommended that you clear the memory before transferring data to the 5500, assuming that you don't need the current contents of the 5500 buffers.

Parameters

None.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration to clear 5500 memory.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 5;  /* memory_clear ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

6 CURSOR_UPDATE_DISABLE

Summary

Command Line:

byte offset	data required	data type
command + 0	cursor_update_disable I.D. (6 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The CURSOR_UPDATE_DISABLE function disables the power spectrum updates on the 5500 that normally occur on the 5500 when the cursors are moved. This is particularly useful when moving a cursor multiple times.

Parameters

None.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration for cursor update disable.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 6;  /* cursor_update_disable ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

7 CURSOR_UPDATE_ENABLE

Summary

Command Line:

byte offset	data required	data type
command + 0	cursor_update_enable I.D. (7 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The CURSOR_UPDATE_ENABLE function re-enables the power spectrum updates normally performed on the 5500 when cursors are moved. It reverses the effect of the CURSOR_UPDATE_DISABLE function.

Parameters

None.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration for cursor update disable.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 7;  /* cursor_update_enable ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                          buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

8 EXPRESS_START

Summary

Command Line:

byte offset	data required	data type
command + 0	express_start I.D. (8 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	screen	unsigned int.

Description

The EXPRESS_START function starts the specified screen(s) on the 5500 scrolling. This function performs the same function as pressing scroll left and scroll right on the front panel or remote keyboard.

Parameter

screen: The screen to scroll: 0 for lower, 1 for upper, 2 for both.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration for express start.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 8;  /* express_start ID */
        buffer[1] = 3;  /* parm count */
        buffer[2] = 0;  /* scroll lower screen */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

9 EXPRESS_STOP

Summary

Command Line:

byte offset	data required	data type
command + 0	express_stop I.D. (9 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	screen	unsigned int.

Description

The EXPRESS_STOP function stops the specified screen(s) on the 5500 express scrolling. This reverses the effect of EXPRESS_START.

Parameter

screen: The screen to stop scrolling: 0 for lower, 1 for upper, 2 for both.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration for express stop.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 9;  /* express_stop ID */
        buffer[1] = 3;  /* parm count */
        buffer[2] = 0;  /* stop lower screen scrolling */

        sendCommand(buffer); /* user program to transmit
                                          buffer */

        UnlUntBus();                      /* user routine */

        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 64)             /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

10 RECALL_SETUP_TABLE

Summary

Command Line:

byte offset	data required	data type
command + 0	recall_setup_table I.D. (10 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	setup	unsigned int.

Description

The RECALL_SETUP_TABLE function recalls the specified stored setup on the 5500. This is equivalent to entering that number on the "ENTRY" keys on the 5500 front panel or remote keyboard.

Parameter

setup: The number of the setup to recall. Must be decimal 0 – –99.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration to recall setup table.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 10; /* recall_setup_table ID */
        buffer[1] = 3;  /* parm count */
        buffer[2] = 32; /* recall user-defined setup number 32 */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

11 WINDOW_ON

Summary

Command Line:

byte offset	data required	data type
command + 0	window_on I.D. (11 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	size	unsigned int.

Data:

byte offset	data required	data type
data + 0	text	string.

Description

The WINDOW_ON function displays the specified text on the lower half of the 5500 screen. This function only displays the number of characters specified in size. Use WINDOW_OFF to remove the window.

Parameters

size: Number of characters in the text.

text: Characters to be displayed.

SRQ response:

Command Line Response:
SRQ #67, "waiting for data," will be issued if call is successful.
Data Response:
SRQ #64 will be issued if data is received successfully.

Example:

Command buffer configuration to display a window of text.

```
main()
{
        int buffer[6];
        char textBuffer[] = "Please stand by...";
        int spollByte;

        buffer[0] = 11; /* window_on ID */
        buffer[1] = 4;  /* parm count */

        buffer[2] = (unsigned)strlen(textBuffer); /* text length */


        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */
```

```
        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 67)              /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* send text location and length to output routine */

        WriteToTranslator(textBuffer, (buffer + 2));    /* user routine */

        UnlUntBus();                      /* user routine */

        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 64)              /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");
}
```

12 WINDOW_OFF

Summary

Command Line:

byte offset	data required	data type
command + 0	window_off I.D. (12 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	size	unsigned int.

Data:

byte offset	data required	data type
data + 0	text	string.

Description

The WINDOW_OFF function turns off a window previously displayed by the WINDOW_ON function. Be sure that the parameters passed to WINDOW_OFF are the same as those passed to WINDOW_ON.

Parameters

size: Number of characters in the text.

text: Characters to be displayed.

SRQ response:

Command Line Response:
SRQ #67, "waiting for data", will be issued if call is successful.
Data Response:
SRQ #64 will be issued if data is received successfully.

Example:

Command buffer configuration to remove a window of text.

```
main()
{
        int buffer[6];
        char textBuffer[] = "Please stand by...";
        int spollByte;

        buffer[0] = 12; /* window_off ID */
        buffer[1] = 4;  /* parm count */

        buffer[2] = (unsigned)strlen(textBuffer); /* text length */


        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */
```

```
        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 67)              /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* send text location and length to output routine */

        WriteToTranslator(textBuffer, (buffer + 2));    /* user routine */

        UnlUntBus();                      /* user routine */

        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 64)              /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

13 GET_5500_VARBLOCK

Summary

Command Line:

byte offset	data required	data type
command + 0	get_5500_varblock I.D. (13 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Data:

byte offset	data returned	data type
data + 0	vars_5500	struct keyvars_5500.

Description

The GET_5500_VARBLOCK function returns a block of information from the 5500 regarding the current status of the 5500 variables defined in a structure of type *KEYVARS_5500*. This structure is defined in appendix D.

Parameter

None.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get 5500 varblock.

```
main()
{
        int buffer[6];
        int spollByte;
        struct keyvars_5500 *keyvarsPtr;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        keyvarsPtr = (struct keyvars_5500*)malloc(sizeof
                        (struct keyvars_5500));


        buffer[0] = 13; /* get_5500_varblock ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                            buffer */
```

```
        UnlUntBus();                        /* user routine */

        spollByte = GetSpollByte()          /* user routine */

        if (spollByte != 66)                /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* Receive data from translator up to sizeof(keyvars_5500) bytes */

        *byteCount = (unsigned)sizeof(struct keyvars_5500);

        ReadFromTranslator((int*)keyvarsPtr, byteCount);        /* user routine */

        UnlUntBus();                        /* user routine */

        spollByte = GetSpollByte()          /* user routine */

        if (spollByte != 64)                /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        /* print out status of 'reconflag' */

        if (keyvarsPtr->reconflag == 1)
           printf("\nRecording.\n");

        else
           printf("\nNot recording.\n");


}
```

14 GET_ANFORM

Summary

Command Line:

byte offset	data required	data type
command + 0	get_anform I.D. (14 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	screen	integer.

Data:

byte offset	data returned	data type
data + 0	analysis format	integer.

Description

GET_ANFORM returns the analysis format of the requested screen. The `#defines` for these format types are listed in appendix A.

Parameter

screen: either UPPER_SCREEN or LOWER_SCREEN. (`#defines` in appendix B).

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get analysis format.

```
main()
{
        int buffer[6];
        int spollByte;
        int dataBuffer[10];
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        buffer[0] = 14;                 /* get_anform ID */
        buffer[1] = 3;                  /* parm count */
        buffer[2] = LOWER_SCREEN;       /* lower screen */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                     /* user routine */

        spollByte = GetSpollByte()       /* user routine */
```

```
if (spollByte != 66)            /* error */
{
   printf("\nError has occured.\n");
   exit(-1);
}

/* read from translator up to 10 bytes */

*byteCount = 10;

ReadFromTranslator(dataBuffer, byteCount);      /* user routine */

UnlUntBus();                    /* user routine */

spollByte = GetSpollByte()      /* user routine */

if (spollByte != 64)            /* error */
{
   printf("\nError has occured.\n");
   exit(-1);
}

printf("\nCall was successful.\n");

switch(*(dataBuffer + 0))
{

  case NODISP_ANFORM:            /* No Display */

        printf("\nNo Display on that screen.\n");
        break;

  case PSPEC_ANFORM:            /* Power Spectrum */
        printf("\nPower spectrum on that screen.\n");
        break;

  case PWRBETC_ANFORM:          /* Power Between Cursors */
        printf("\nPower between cursors on that screen.\n");
        break;

  case PWRATC_ANFORM:           /* Power At Cursors */
        printf("\nPower At cursors on that screen.\n");
        break;

  case MINISON_ANFORM:          /* Minison Display */
        printf("\nMinison display on that screen.\n");
        break;

  case WATER_ANFORM:            /* Waterfall */
        printf("\nWaterfall display on that screen.\n");
        break;

  case COMB_ANFORM:             /* Combination */
```

```
            printf("\nCombination display on that screen.\n");
            break;

        case AMP_ANFORM:              /* Amplitude */
            printf("\nAmplitude display on that screen.\n");
            break;

        case WAVE_ANFORM:             /* Waveform  */
            printf("\nWaveform display on that screen.\n");
            break;

        case SONAGRAM_ANFORM:         /* Sonagram (TM) */
            printf("\nSonagram display on that screen.\n");
            break;

        case LTSA_ANFORM:             /* Long-Term Spectral Average */
            printf("\nLong term spectral average on that screen.\n");
            break;


    } /* end of cases */

}
```

15 GET_SCREEN_CHANNEL

Summary

Command Line:

byte offset	data required	data type
command + 0	get_screen_channel I.D. (15 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	screen	integer.

Data:

byte offset	data returned	data type
data + 0	channel	integer.

Description

GET_SCREEN_CHANNEL returns the channel being displayed on the requested screen.

Parameter

screen: either UPPER_SCREEN or LOWER_SCREEN. (**#defines** in appendix B).

SRQ response:

Command Line Response:
SRQ #66, "data to send", will be issued if call is successful.
Data Response:
SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get screen channel.

```
main()
{
        int buffer[6];
        int spollByte;
        int dataBuffer[10];
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        buffer[0] = 15;          /* get_screen_channel ID */
        buffer[1] = 3;                   /* parm count */
        buffer[2] = LOWER_SCREEN;        /* lower screen */

        sendCommand(buffer); /* user program to transmit
                                          buffer */

        UnlUntBus();                      /* user routine */

        spollByte = GetSpollByte()        /* user routine */
```

```
        if (spollByte != 66)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to 10 bytes */

        *byteCount = 10;

        ReadFromTranslator(dataBuffer, byteCount);      /* user routine */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        if (*(dataBuffer + 0) == 1)
           printf("\nChannel ONE being displayed.\n");

        else
           printf("\nChannel TWO being displayed.\n");
}
```

16 BUFFER_SIZE

Summary

Command Line:

byte offset	data required	data type
command + 0	buffer_size I.D. (16 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	channel	integer.

Data:

byte offset	data returned	data type
data + 0	samples	unsigned long int.

Description

BUFFER_SIZE returns the the number of samples are in the specified buffer (channel) in the 5500.

Parameter

channel: 1 for Channel 1, 2 for Channel 2.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get buffer size.

```
main()
{
        int buffer[6];
        int spollByte;
        int dataBuffer[10];
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        buffer[0] = 16;                 /* buffer_size ID */
        buffer[1] = 3;                  /* parm count */
        buffer[2] = 1;                  /* lower screen */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                     /* user routine */

        spollByte = GetSpollByte()       /* user routine */
```

```
        if (spollByte != 66)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to 10 bytes */

        *byteCount = 10;

        ReadFromTranslator(dataBuffer, byteCount);      /* user routine */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        printf("\nThe number of samples is: %lu.",
                       *(long unsigned*)dataBuffer);

}
```

17 GET_CURS_VALS

Summary

Command Line:

byte offset	data required	data type
command + 0	get_curs_vals I.D. (17 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Data:

byte offset	data returned	data type
data + 0	curs_vals	struct CURSOR_VALUES.

Description

The GET_CURS_VALS function returns a block of information from the 5500 regarding the 5500 cursor variables defined in a structure of type *CURSOR_VALUES*. This structure is defined in appendix D.

Parameter

None.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get 5500 varblock.

```
main()
{
        int buffer[6];
        char printBuffer[20];
        int spollByte;
        struct CURSOR_VALUES *cursvalPtr;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        cursvalPtr = (struct CURSOR_VALUES*)malloc(sizeof
                        (struct CURSOR_VALUES));


        buffer[0] = 17; /* get_curs_vals ID */
        buffer[1] = 2;  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */
```

```
        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 66)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to sizeof(CURSOR_VALUES) bytes */

        *byteCount = (unsigned)sizeof(struct CURSOR_VALUES);

        ReadFromTranslator((int*)cursvalPtr, byteCount);  /* user routine */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        /* print out some cursor information */

        /* first, copy it into output buffer and terminate string */

        strncpy(printBuffer, cursvalPtr->deltat, 17);
        printBuffer[17] = '\0';

        /* print out time between cursors */

        printf("\nTime between cursors is: %s\n", printBuffer);

}
```

18 GET_TIME_SLICE

Summary

Command Line:

byte offset	data required	data type
command + 0	get_time_slice ID (18 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	source	integer.

Data:

byte offset	data returned	data type
data + 0	spectrum	unsigned int.

Description

The GET_TIME_SLICE function returns a block of spectral data from the 5500. The type of the data is defined in appendix NNN.

Parameter

source: Where to get data from in the 5500. Possible values are UPPER_SCREEN, LOWER_SCREEN, and RED_CURSOR.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get spectral data.

```
main()
{
        int buffer[6];
        char printBuffer[20];
        int spollByte;
        int *spectralData;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));

        spectralData = (int*)malloc(2048 * sizeof(int));

        buffer[0] = 18;          /* get_time_slice ID */
        buffer[1] = 3;           /* parm count */
        buffer[2] = LOWER_SCREEN; /* from lower screen */

        sendCommand(buffer); /* user program to transmit
                                        buffer */
```

```
        UnlUntBus();                        /* user routine */

        spollByte = GetSpollByte()          /* user routine */

        if (spollByte != 66)                /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to 2048 bytes */

        *byteCount = 2048;

        ReadFromTranslator(spectralData, byteCount);    /* user routine */

        UnlUntBus();                        /* user routine */

        spollByte = GetSpollByte()          /* user routine */

        if (spollByte != 64)                /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

19 GET_VCBFLAG

Summary

Command Line:

byte offset	data required	data type
command + 0	get_vcbflag I.D. (19 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Data:

byte offset	data returned	data type
data + 0	screen	integer.

Description

The GET_VCBFLAG function returns the location of the time cursors on the 5500. This call does not determine whether or not the time cursors are displayed; it only indicates where they would be if they were displayed. The status of the time cursors (displayed / not displayed) can be determined by a call to GET_5500_VARBLOCK.

Parameter

None.

SRQ response:

Command Line Response:
SRQ #66, "data to send", will be issued if call is successful.
Data Response:
SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get vcbflag.

```
main()
{
        int buffer[6];
        int dataBuffer[10];
        int spollByte;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        buffer[0] = 19;          /* get_vcbflag ID */
        buffer[1] = 2;           /* parm count */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */
```

```
        if (spollByte != 66)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to 10 bytes */

        *byteCount = 10;

        ReadFromTranslator(dataBuffer, byteCount);      /* user routine */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        if (*(dataBuffer + 0) == LOWER_SCREEN)
          printf("\nThe cursors are on the lower screen.\n");

        else
           printf("\nThe cursors are on the upper screen.\n");

}
```

20 GET_BUFFER

Summary

Command Line:

byte offset	data required	data type
command + 0	get_buffer I.D. (20 dec)	integer.
command + 2	parameter count (5 dec)	integer.
command + 4	source	integer.
command + 6	numbits	integer.
command + 8	neg_flag	integer.

Data:

byte offset	data returned	data type
data + 0	acoustical_data	***

*** The size of the individual data samples is dependent on the resolution asked for. 8 bits uses up one byte, all others require two bytes.

Description

The GET_BUFFER function returns a data buffer from the 5500. This is a LARGE (MULTIPLE) data transfer call, therefore data buffers larger than 64K will be transferred in blocks of FFF0 hex (64K - 16) bytes. The user program will have to deal with these large data blocks, as well as the handshaking via the SRQ bytes. (It is not guaranteed that all data blocks will be FFF0 hex bytes. The user should keep track of the byte count, and like all data transfers, use the EOI line to detect the end of a block.)

Parameters

source: Where to get the data from in the 5500. Possible values are UPPER_SCREEN, LOWER_SCREEN, BETWEEN_CURSORS, CHANNEL_1, CHANNEL_2.

numbits: The number of bits of resolution (using only 8 bits will reduce total buffer size by 1/2).

neg_flag: This flag indicates how to handle the sign of the data. If TRUE, the data will be inverted as it is transferred. If FALSE, it will maintain the sign it has in the 5500.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is initially successful and everytime there is still data left to send.

Data Response:

SRQ #64 will be issued if all data was sent successfully.

Example:

Command buffer configuration to get buffer.

```
main()
{
        int buffer[6];
```

```
int *dataBuffer;
int spollByte;
int endOfLoopSet = 0;
unsigned *byteCount;

byteCount = (unsigned *)malloc(sizeof(unsigned));


dataBuffer = (int*)halloc(65536, sizeof(char));

buffer[0] = 22;          /* get_buffer ID */
buffer[1] = 5;           /* parm count */
buffer[2] = CHANNEL_1;   /* fron channel one */
buffer[3] = 8;           /* use 8 bits */
buffer[4] = FALSE;       /* do not invert the data */

sendCommand(buffer); /* user program to transmit

UnlUntBus();                      /* user routine */

spollByte = GetSpollByte()        /* user routine */

if (spollByte != 66)             /* error */
{
   printf("\nError has occured.\n");
   exit(-1);
}

while (endOfLoopSet != 1)
{
  /* Read in a maximum of 65520 bytes at a time */

  *byteCount = 0xFFF0;

  ReadFromTranslator(dataBuffer, byteCount);    /* user routine */

  /* Write the new data to a file */

  SpoolDataToFile(dataBuffer, byteCount);       /* user routine */

  UnlUntBus();                   /* user routine */

  spollByte = GetSpollByte()     /* user routine */


  if (spollByte == 66)
    printf("\nMore data to go.\n");

  else if (spollByte == 64)
  {
    printf("\nAll data received OK.\n");
    endOfLoopSet = 1;
  }
```

```
        else
        {
          printf("\nThere was an error.\n");
          endOfLoopSet = 1;
        }

      } /* end of while */


}
```

21 SEND_BUFFER

Summary

Command Line:

byte offset	data required	data type
command + 0	send_buffer I.D. (21 dec)	integer.
command + 2	parameter count (7 dec)	integer.
command + 4	length of buffer	long unsigned int.
command + 6	destination	integer.
command + 8	numbits	integer.
command + 10	neg_flag	integer.

Data:

byte offset	data to send	data type
data + 0	acoustical_data	***

*** The size of the individual data samples is dependent on the resolution asked for. 8 bits uses up one byte, all others require two bytes.

Description

The SEND_BUFFER function sends a data buffer to the 5500. This is a LARGE (MULTIPLE) data transfer call, therefore data buffers larger than 64K must be transferred in blocks no larger than FFF0 hex (64K - 16) bytes at a time, until completion of the transfer. It is important to send exactly as many bytes as was called out in the command line parameter "length of buffer," otherwise the translator will view this as an error. As with all data transfers, each block of data should be terminated by asserting the EOI line.

Parameters

length of buffer: The total number of BYTES in the buffer to be sent.

source: Where to put the data in the 5500. Possible values are RED_CURSOR (to begin transferring data at the red cursor), and CHANNEL_1 or CHANNEL_2. There is one difference between these choices; CHANNEL_1 and CHANNEL_2 will update the screen display(s) of the appropriate channel so that the left edge of the screen will be aligned with the start of the data just transferred.

numbits: The number of bits of resolution. (using only 8 bits will reduce total buffer size by 1/2.)

neg_flag: This flag indicates how to handle the sign of the data. If TRUE, the data will be inverted as it is transferred. If FALSE, it will maintain the sign it has in the 5500.

SRQ response:

Command Line Response:

SRQ #67, "waiting for data", will be issued if call is initially successful and everytime there is more data expected.

Data Response:

SRQ #64 will be issued if all data was received successfully.

Example:

Command buffer configuration to send buffer.

```
main()
{
        int buffer[6];
        int *dataBuffer;
        int spollByte;
        int endOfLoopSet = 0;
        long unsigned int sizeOfFile;
        unsigned bytesThisTime;

        struct SendBufferBlock
        {
          int idByte;
          int parmCount;
          long unsigned int totalByteCount;
          int destination;
          int numbits;
          int neg_flag;
        };

        struct SendBufferBlock *sbPtr;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        sbPtr = (struct SendBufferBlock*)malloc(sizeof
                (struct SendBufferBlock));

        dataBuffer = (int*)halloc(65536, sizeof(char));


        sizeOfFile = GetFileSize()  /* user routine */

        sbPtr->idByte = 23;                     /* send buffer ID */
        sbPtr->parmCount = 7;                   /* parm count */
        sbPtr->totalByteCount = sizeOfBuffer;   /* number of bytes */
        sbPtr->destination = CHANNEL_1;         /* to channel 1 */
        sbPtr->numbits = 8;                     /* use 8 bit res. */
        sbPtr->neg_flag = FALSE;                /* do not invert */

        sendCommand((int*)sbPtr); /* user program to transmit

        UnlUntBus();                        /* user routine */

        spollByte = GetSpollByte()          /* user routine */

        if (spollByte != 67)                /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }
```

```
    while (endOfLoopSet != 1)
    {

      /* load in up to 65520 bytes from file */

      bytesThisTime = FillBufferFromFile(dataBuffer, 0xFFF0);

      /* write to translator number of bytes received from file */

      WriteToTranslator(dataBuffer, &bytesThisTime);

      UnlUntBus();                  /* user routine */

      spollByte = GetSpollByte()    /* user routine */


      if (spollByte == 67)
        printf("\nSending more data.\n");

      else if (spollByte == 64)
      {
        printf("\nAll sent received OK.\n");
        endOfLoopSet = 1;
      }

      else
      {
        printf("\nThere was an error.\n");
        endOfLoopSet = 1;
      }

    } /* end of while */


}
```

22 SCSI_ERROR

Summary

Command Line:

byte offset	data required	data type
command + 0	scsi_error I.D. (22 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Data:

byte offset	data returned	data type
data + 0	error_text	string.

Description

SCSI_ERROR returns a string of text which describes the result of the most recent SCSI call made by the translator. If the translator ever returns a #65 SRQ byte for any call, this call will retrieve specific information about the error.

Parameter

None.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get SCSI error.

```
main()
{
        int buffer[6];
        int dataBuffer[50];
        int spollByte;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        buffer[0] = 24;          /* scsi_error ID */
        buffer[1] = 2;           /* parm count */

        sendCommand(buffer); /* user program to transmit
                                          buffer */

        UnlUntBus();                      /* user routine */

        spollByte = GetSpollByte()        /* user routine */
```

```
        if (spollByte != 66)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to 50 bytes */

        *byteCount = 50;

        ReadFromTranslator(dataBuffer, byteCount);      /* user routine */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        printf("\nSCSI message: %s.\n", (char*)dataBuffer);


}
```

23 GET_SAMP_FREQ

Summary

Command Line:

byte offset	data required	data type
command + 0	get_samp_freq I.D. (23 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	channel	integer.

Data:

byte offset	data returned	data type
data + 0	sampling_frequency	long int.

Description

GET_SAMP_FREQ returns the sampling frequency of a specified buffer. Note that the FREQUENCY RANGE of a channel is lower than the SAMPLING FREQUENCY by a factor of 2.56. (SAMP_FREQ / 2.56 = FREQ_RANGE).

Parameter

channel: 1 for channel one, 2 for channel two.

SRQ response:

Command Line Response:

SRQ #66, "data to send", will be issued if call is successful.

Data Response:

SRQ #64 will be issued if data is sent successfully.

Example:

Command buffer configuration to get sampling frequency.

```
main()
{
        int buffer[6];
        int dataBuffer[10];
        int spollByte;
        unsigned *byteCount;

        byteCount = (unsigned *)malloc(sizeof(unsigned));


        buffer[0] = 25;          /* get_samp_freq ID */
        buffer[1] = 3;           /* parm count */
        buffer[2] = 1;           /* channel one */

        sendCommand(buffer); /* user program to transmit
                                          buffer */

        UnlUntBus();                      /* user routine */
```

```
        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 66)              /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        /* read from translator up to 10 bytes */

        *byteCount = 10;

        ReadFromTranslator(dataBuffer, byteCount);      /* user routine */

        UnlUntBus();                      /* user routine */

        spollByte = GetSpollByte()        /* user routine */

        if (spollByte != 64)              /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

        printf("\nSampling frequency is: %ld samples per second.\n",
                 *(long int*)dataBuffer);

        printf("\nThe Frequency Range is: %ld hz.\n",
                (long int)(*(long int*)dataBuffer / 2.56));

}
```

24 CHANGE_IEEE_ADDRESS

Summary

Command Line:

byte offset	data required	data type
command + 0	change_ieee_address I.D. (24 dec)	integer.
command + 2	parameter count (3 dec)	integer.
command + 4	address	integer.

Description

The CHANGE_IEEE_ADDRESS function allows the active controller to change the translator's default device address of '22' to another valid IEEE device address.

Parameter

address: check on this...

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration for change ieee address.

```
main()
{
        int buffer[6];
        int spollByte;


        buffer[0] = 26; /* change ieee address ID */
        buffer[1] = 3;  /* parm count */
        buffer[2] = 23; /* device address #23 */

        sendCommand(buffer); /* user program to transmit
                                        buffer */

        UnlUntBus();                    /* user routine */

        spollByte = GetSpollByte()      /* user routine */

        if (spollByte != 64)           /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

25 INIT_TRANS

Summary

Command Line:

byte offset	data required	data type
command + 0	init_trans I.D. (25 dec)	integer.
command + 2	parameter count (2 dec)	integer.

Description

The INIT_TRANS function "byte aligns" data in and out of the translator to be correct for any computer. Make this call before attempting any other calls.

Parameters

none.

SRQ response:

SRQ #64 will be issued if call is successful.

Example:

Command buffer configuration for init trans.

```
main()
{
        int buffer[6];
        int *dataBuffer;
        int spollByte;

        buffer[0] = 27;                 /* init trans ID */
        buffer[1] = 2;                  /* parm count */

        sendCommand(buffer); /* user program to transmit
                                            buffer */

        UnlUntBus();                     /* user routine */

        spollByte = GetSpollByte()       /* user routine */

        if (spollByte != 64)            /* error */
        {
           printf("\nError has occured.\n");
           exit(-1);
        }

        printf("\nCall was successful.\n");

}
```

A DEFINES FOR ANALYSIS FORMAT TYPES

```
#define NODISP_ANFORM    0x00    /* no display */
#define PSPEC_ANFORM     0x07    /* power spectrum */
#define PWRBETC_ANFORM   0x0A    /* power between cursors */
#define PWRATC_ANFORM    0x03    /* Power at cursors */
#define MINISON_ANFORM   0x20    /* minison display */
#define WATER_ANFORM     0x10    /* waterfall */
#define COMB_ANFORM      0x40    /* combination */
#define AMP_ANFORM       0x80    /* amplitude */
#define WAVE_ANFORM      0x02    /* waveform */
#define SONAGRAM_ANFORM  0x04    /* Sonagram (tm) */
#define LTSA_ANFORM      0x0B    /* long term spectral average */
```

B DEFINES FOR OTHER CONSTANTS

```
#define UPPER_SCREEN     0
#define LOWER_SCREEN     1
#define RED_CURSOR       2
#define BETWEEN_CURSORS  3
#define CHANNEL_1        4
#define CHANNEL_2        5
```

C DEFINES FOR 5500 KEY CODES

```
#define CH1_10_DB_DOWN_          0      /* CH1 10 DB DOWN*/
#define CH1_1_DB_DOWN_           1      /* CH1 1 DB DOWN*/
#define CH1_COUPLING_            2      /* CH1 COUPLING*/
#define CH2_COUPLING__           3      /* CH2 COUPLING*/
#define CH2_10_DB_DOWN_          4      /* CH2 10 DB DOWN*/
#define CH2_1_DB_DOWN_           5      /* CH2 1 DB DOWN*/
#define CH2_10_DB_UP_            6      /* CH2 10 DB UP*/
#define CH2_1_DB_UP_             7      /* CH2 1 DB UP*/
#define CH1_VOL_DOWN_            8      /* CHANNEL 1 VOLUME DOWN*/
#define CH1_VOL_UP_              9      /* CHANNEL 1 VOLUME UP*/
#define CH2_VOL_UP_              10     /* CHANNEL 2 VOLUME UP*/
#define RECORD_OFF_              11     /* RECORD OFF*/
#define STOP_                    12     /* STOP*/
#define CH2_VOL_DOWN_            13     /* CHANNEL 2 VOLUME DOWN*/
#define RECORD_ON_               14     /* RECORD ON*/
#define TIME_CURSOR_LEFT_        16     /* TIME CURSOR LEFT*/
#define TIME_CURSOR_RIGHT_       17     /* TIME CURSOR RIGHT*/
#define FREQUENCY_CURSOR_STORE_  18     /* FREQUENCY CURSOR STORE*/
#define FREQUENCY_CURSOR_UP_     19     /* FREQUENCY CURSOR UP*/
#define FREQUECY_CURSOR_CLEAR_   20     /* FREQUECY CURSOR CLEAR*/
#define FREQUENCY_CURSOR_DOWN_   21     /* FREQUENCY CURSOR DOWN*/
#define SCROLL_UP_LEFT_          22     /* SCROLL UP LEFT*/
#define SCROLL_LOW_LEFT_         23     /* SCROLL LOW LEFT*/
#define PRINT_                   25     /* PRINT*/
#define FUNCTION_A_              26     /* FUNCTION A*/
#define FUNCTION_B_              27     /* FUNCTION B*/
#define FUNCTION_C_              28     /* FUNCTION C*/
#define FUNCTION_D_              29     /* FUNCTION D*/
#define FUNCTION_E_              30     /* FUNCTION E*/
#define FUNCTION_F_              31     /* FUNCTION F*/
#define FREQUENCY_CURSOR_ONOFF_  32     /* FREQUENCY CURSOR ON/OFF*/
#define CH1_1_DB_UP_             34     /* CH1 1 DB UP*/
#define CH1_10_DB_UP_            35     /* CH1 10 DB UP*/
#define TIME_CURSOR_UPPERLOWER_  36     /* TIME CURSOR UPPER/LOWER*/
#define TIME_CURSOR_ONOFF_       37     /* TIME CURSOR ON/OFF*/
#define TIME_CURSOR_STORE_       38     /* TIME CURSOR STORE*/
#define TIME_CURSOR_CLEAR_       39     /* TIME CURSOR CLEAR*/
#define MENU_SELECT_             40     /* MENU SELECT*/
#define CH2_MUTE_                41     /* CHANNEL 2 MUTE*/
#define MENU_POS_UP_             42     /* MENU POSITION UP*/
#define MENU_ON_OFF_             43     /* MENU ON/OFF*/
#define BUTTON_9_                44     /* BUTTON_9*/
#define BUTTON_7_                45     /* BUTTON_7*/
#define MENU_POS_L_R_            46     /* MENU POSITION LEFT/RIGHT*/
#define MENU_POS_DOWN_           47     /* MENU POSITION DOWN*/
#define DECIMAL_POINT_           48     /* DECIMAL POINT*/
#define BUTTON_1_                49     /* BUTTON_1*/
#define MONO_STEREO_             50     /* MONO/STEREO*/
```

```
#define CH1_MUTE_              51      /* CHANNEL 1 MUTE*/
#define SCROLL_BOTH_RIGHT_     52      /* SCROLL BOTH RIGHT*/
#define SCROLL_BOTH_LEFT_      53      /* SCROLL BOTH LEFT*/
#define SCROLL_UP_RIGHT_       54      /* SCROLL UP RIGHT*/
#define SCROLL_LOW_RIGHT_      55      /* SCROLL LOW RIGHT*/
#define ENTER_                 56      /* ENTER*/
#define BUTTON_0_              57      /* BUTTON_0*/
#define BUTTON_2_              58      /* BUTTON_2*/
#define BUTTON_3_              59      /* BUTTON_3*/
#define BUTTON_4_              60      /* BUTTON_4*/
#define BUTTON_5_              61      /* BUTTON_5*/
#define BUTTON_6_              62      /* BUTTON_6*/
#define BUTTON_8_              63      /* BUTTON_8*/
```

D DEFINES OF STRUCTURES USED BY THE TRANSLATOR

Descriptions of the fields in this structure are in Appendix E.

```
struct CURSOR_VALUES
{
  char fc1hz[12],  /* FREQUENCY LOCATION OF FC1 */
  fc2hz[12],       /* FREQUENCY LOCATION OF FC2 */
  deltafhz[12],    /* FREQ. DIFFERENCE OF FC'S */
  deltat[17],      /* TIME DIFFERENCE OF TC'S*/
  fc1db[4],        /* FC1 dB LEVEL */
  fc2db[4],        /* FC2 dB LEVEL */
  deltafdb[4],     /* dB LEVEL DIFFERENCE */
  deltar1[17],     /* TIME LOCATION OF CURSOR 1 */
  deltar2[17],     /* TIME LOCATION OF CURSOR 2 */
  pitchtc1[5],     /* PITCH AT TIME CURSOR 1 */
  pitchtc2[5],     /* PITCH AT TIME CURSOR 2 */
  amptc1[5],       /* AMPLITUDE AT TIME CURSOR 1 */
  amptc2[5],       /* AMPLITUDE AT TIME CURSOR 2 */
  jitter[6],       /* JITTER (PERCENT) */
  shimmer[6],      /* SHIMMER (PERCENT) */
  pitchav[7],      /* AVERAGE PITCH (HZ) */
  h2noise[5],      /* HARMONIC TO NOISE (dB) */
  breath[8],       /* BREATHINESS */
  not_used[105];   /* 256 BYTE STRUCTURE */

};


struct keyvars_5500
{
  unsigned char ch1atten[2];
  unsigned char ch1coupling;
  unsigned char ch2atten[2];
  unsigned char ch2coupling;
  unsigned char timecursonflag; /* 0=no 1=yes the time cursors are on */
  unsigned char freqcursonflag; /* 0=no 1=yes the freq cursors are on */
  unsigned char mutes;          /* Bit 0 : 0=no 1=yes ch1 is muted
                                   Bit 1 : 0=no 1=yes ch2 is muted */
  unsigned char menuonflag;     /* 0=no 1=yes menus are on */
  unsigned char menumode;       /* tells if we are in sub-menu or settings display */
  unsigned char leftcolflag;    /* tells which column's settings we are changing */
  unsigned char printingflag;   /* 0=no 1=yes the 5500 is printing */
  unsigned char reconflag;      /* 0=no 1=yes the 5500 is recording */
  union                         /* pointer to TEXTRAMCOPY/SFRRAMCOPY */
  {
    unsigned long addr;
    unsigned char bytes[4];
```

```
  } textramcopy, sfrramcopy;
};
```

E CURSOR DATA INFORMATION

The GET_CURS_VALS call returns its data in a structure. This structure type is defined in the Appendix D, with the designator *CURSOR_VALUES*. The values that are returned in this structure are ASCII text, but not strings since they are not null terminated.

Please note that some of the fields below refer to "error" values for that field. This does not necessarily mean that they indicate an error, only that the specified field is not applicable to the current set-up on the 5500.

fc1hz (12 bytes) This is the frequency at which the red frequency cursor is set. If there is no red frequency cursor, the value will be -1. The units are Hz.

fc2hz (12 bytes) This is the frequency at which the green frequency cursor is set. If there is no green frequency cursor, the value will be -1. The units are Hz.

deltafhz (12 bytes) This is the difference in the frequencies at the two frequency cursors. fc1hz and fc2hz. The units are Hz.

deltat (17 bytes) This is the difference in the time locations of the two time cursors, deltar1 and deltar2. If there aren't two time cursors displayed, the value in this field will be -1.

fc1db (4 bytes) This is the amplitude at the red frequency cursor (for example, on a power spectrum display). The units are in decibels, referenced as on the 5500 screen. The 5500's default for the reference point is 1V rms. If the frequency cursors aren't displayed, or if the display is not appropriate, this field will contain "9999".

fc2db (4 bytes) same as fc1db, but for the green frequency cursor.

deltafdb (4 bytes) The difference between fc1db and fc2db, measured in dB. Again, the error value is "9999".

deltar1 (17 bytes) The location of the red time cursor, measured in seconds and referenced to the oldest point in the buffer. If the time cursors aren't displayed, this value will be -1.

deltar2 (17 bytes) The location of the green time cursor, measured in seconds and referenced to the oldest point in the buffer. If the time cursors aren't displayed, this value will be -1.

pitchc1 (5 bytes) The pitch, in Hz., at the red time cursor. This comes from the combination display. If the pitch isn't available, the value will be -1.

pitchc2 (5 bytes) The pitch, in Hz., at the green time cursor. This comes from the combination display. If the pitch isn't available, the value will be -1.

amptc1 (5 bytes) The amplitude in dB, at the red time cursor (on an amplitude display). -1 indicates an error value.

amptc2 (5 bytes) The amplitude in dB, at the green time cursor (on an amplitude display). -1 indicates an error value.

jitter (6 bytes) The jitter as calculated by the voicing parameters program built into the 5500. The program to make these calculations is Option 1 of the 'MISCELLANEOUS' choice on the 5500's 'OTHER' menu (it may not be available on some older 5500's). The values must be visible on the 5500 screen in order to import them. If the jitter is not available, the value will be -1.0.

shimmer (6 bytes) Another voicing parameter. -1.0 is the error value.

pitchav (7 bytes) Average pitch, measured in Hz.; another voicing parameter. Error is -1.0.

h2noise (5 bytes) In dB. Harmonic to noise ratio; another voicing parameter. 99.9 is the error value.

breath (8 bytes) Breathiness, another voicing parameter. 0.000 is the error value.

not_used (105 bytes) This data space is not used. It pads out the CURSOR_VALUES buffer to 256 bytes. This space may be used in the future.

F SCSI ERROR CODES

The following is a list of the error codes that can be generated by the Kay SCSI functions. The SCSI_ERROR call will return these errors as strings.

0	SCSI card not installed.
1	Transfer completed successfully.
80	DMA transfer buffer crossed 64K boundary.
81	Unable to obtain SCSI bus.
82	Target selection time-out.
83	SCSI bus reset sent/received.
84	Unexpected disconnected/invalid bus phase.
85	Sense data operation failed.
86	Uncorrectable parity error detected.
87	SCSI message data not sent.
88	Message reject sent/received.
89	Invalid driver command code.
8A	Diagnostic failure.
FF	BIOS disk I/O handler SCSI sense data available.
Any other code	Undefined error on SCSI.

G Kay File Format

The following is the definition of the "5500" file format used by the Kay File Management program and by the *get_curs_vals()* function in the 5500 SCSI C Library. File offsets are in bytes.

Offset	Contents
0–37	Unused.
38–39	(word) 0
40–63	Unused.
64–69	"5500SD"
70–73	(long) 1
74–77	(long) 0
78–119	Unused.
120–121	Exponent for sample frequency (= 1 for 5500 data)
122–123	Mantissa for sample frequency
124–125	(word) -32000
126–127	(word) 32149
128–129	0 if sampled data, else spectral data

The following fields have meaning only when the file contains spectral data:

130–131	0 = no else yes the spectrum is 8:1 zoom
132–133	Scale magnification factor
134–135	0 = no else yes exponential averaging is active
136–137	Zoom center bin #
138–139	Scale magnification base bin #
140–141	Averaging factor
142–143	# of bins per spectrum (200 or 400)
144–145	transform window size
146–147	# of spectra in file
148–149	# of points window advance per spectrum.

The rest of the file has meaning for sampled data and spectra:

150–511	Unused.
512–end	Data

In addition, the following fields are inserted for sampled data as in the MSL format:

26–27	Number of bits (two characters, ASCII form)
28–35	Number of data points (6 or 7 character, null terminated, ASCII string)

Index

address, bus iv, 42
ALL_KEYS_UP 3
analyzed data 28
BUFFER_SIZE 24
CHANGE_IEEE_ADDRESS 42
Command Lines
    general description iv
    parameter count iv
CURSOR DATA INFORMATION 49
cursor location 30
cursor values 26, 47
CURSOR_UPDATE_DISABLE 8
CURSOR_UPDATE_ENABLE 9
DEFINES FOR ANALYSIS FORMAT TYPES 44
DEFINES FOR OTHER CONSTANTS 44
EOI iv
Error codes, SCSI 38, 51
Errors, types of iv
EXPRESS_START 10
EXPRESS_STOP 11
File Format, "Kay" 52
Flow Charts, reference to v
format, analysis 19 subitem defines 44
GET_5500_VARBLOCK 17
GET_ANFORM 19
GET_BUFFER 32
GET_CURS_VALS 26
GET_SAMP_FREQ 40
GET_SCREEN_CHANNEL 22
GET_TIME_SLICE 28
GET_VCBFLAG 30
INIT_TRANS 43
key codes, definitions 45
key codes, sending 1
keyboard functions 1
KEY_DOWN 1
LOCK_5500_KBD 5
MEMORY_CLEAR 7
panel functions 1
Possible Calls, summary of vi
RECALL_SETUP_TABLE 12
sampled data 32
sampled data 35
sampling frequency 40
scrolling, express 10, 11
SCSI_ERROR 38
SEND_BUFFER 35
Service Request Code, possible values viii
Service Requests iv
spectrum data 28
SRQ viii
structure definitions 47
UNLOCK_5500_KBD 6
WINDOW_OFF 15
WINDOW_ON 13

```
#include "trans.h"

/* This program simulates an IEEE488 user making calls to
   our IEEE translator. The IO-TECH card is initialized,
   then "calls" are made by sending out the appropriate
   data on the ieeebus */


int transAddress = 22;
int printEnableSet = 0;

main()
{
   char response;
   char *menu_choice;
   int *dataBuffer;


   if (ieeeinit() != -1)
   {
      dataBuffer = (int *)halloc(65536,sizeof(char));

      if (dataBuffer == NULL)
         printf("\nCalloc trouble");


      ieeewt("EOL OUT LF\r\n");


      /* At this point, the IO-Tech card has been initialized.
         A menu will prompt the user to press a specified key
         for a given function, "M" to see the menu again,
         or "Q" to quit. */


         menu_choice = &response;

         *menu_choice = 'm';

         printMenu();

         while ((*menu_choice != 'Q')&&(*menu_choice != 'q'))
         {
            scanf("%1s", menu_choice);


            if ((*menu_choice == 'A')||(*menu_choice == 'a'))
               Clear5500Memory(dataBuffer);

            else if ((*menu_choice == 'B')||(*menu_choice == 'b'))
               DisableCursorUpdates(dataBuffer);

            else if ((*menu_choice == 'C')||(*menu_choice == 'c'))
               GetCursorValues(dataBuffer);

```

```
56          else if ((*menu_choice == 'D')||(*menu_choice == 'd'))
57             EnableCursorUpdates(dataBuffer);
58
59          else if ((*menu_choice == 'E')||(*menu_choice == 'e'))
60             TogglePrintSwitch();
61
62          else if ((*menu_choice == 'F')||(*menu_choice == 'f'))
63             ExpressStart(dataBuffer);
64
65          else if ((*menu_choice == 'G')||(*menu_choice == 'g'))
66             ExpressStop(dataBuffer);
67
68          else if ((*menu_choice == 'H')||(*menu_choice == 'h'))
69             RecallSetupTable(dataBuffer);
70
71          else if ((*menu_choice == 'I')||(*menu_choice == 'i'))
72             Get5500Varblock(dataBuffer);
73
74          else if ((*menu_choice == 'J')||(*menu_choice == 'j'))
75             GetAnform(dataBuffer);
76
77          else if ((*menu_choice == 'K')||(*menu_choice == 'k'))
78             GetScreenChannel(dataBuffer);
79
80          else if ((*menu_choice == 'L')||(*menu_choice == 'l'))
81             LockAndUnlockKeyboard(dataBuffer);
82
83          else if ((*menu_choice == 'M')||(*menu_choice == 'm'))
84             printMenu();
85
86          else if ((*menu_choice == 'N')||(*menu_choice == 'n'))
87             GetBufferSize(dataBuffer);
88
89          else if ((*menu_choice == 'O')||(*menu_choice == 'o'))
90             GetCursorLocation(dataBuffer);
91
92          else if ((*menu_choice == 'P')||(*menu_choice == 'p'))
93             GetScsiError(dataBuffer);
94
95          else if ((*menu_choice == 'T')||(*menu_choice == 't'))
96             GetSamplingFrequency(dataBuffer);
97
98          else if ((*menu_choice == 'R') || (*menu_choice == 'r'))
99             PressRecordButton(dataBuffer);
100
101         else if ((*menu_choice == 'S')||(*menu_choice == 's'))
102            PressStopButton(dataBuffer);
103
104         else if ((*menu_choice == 'U')||(*menu_choice == 'u'))
105            SendBuffer(dataBuffer);
106
107         else if ((*menu_choice == 'V')||(*menu_choice == 'v'))
108            InitTrans(dataBuffer);
109
110         else if ((*menu_choice == 'X')||(*menu_choice == 'x'))
```

```
111                ChangeIeeeAddress(dataBuffer);
112
113             else if ((*menu_choice == 'Y')||(*menu_choice == 'y'))
114                GetBuffer(dataBuffer);
115
116             else if ((*menu_choice == 'W')||(*menu_choice == 'W'))
117                PutUpAWindow(dataBuffer);
118
119             else if ((*menu_choice == 'Z')||(*menu_choice == 'z'))
120                GetTimeSlice(dataBuffer);
121
122             else if ((*menu_choice == '1')||(*menu_choice == '1'))
123                PressAnyKey(dataBuffer);
124
125             else if ((*menu_choice == '2')||(*menu_choice == '2'))
126                TransExit(dataBuffer);
127
128
129
130
131          }
132
133       hfree(dataBuffer);
134
135       printf("\nProgram ended.\n");
136
137
138
139    }
140
141    else
142    {
143       printf("\nCould not initialize IO-Tech card.\n");
144    }
145
146
147
148 }
149 printMenu()
150 {
151
152    printf("\nMenu Selections:\n");
153    printf("<1> to PRESS ANY KEY\t\t<O> to GET CURSOR LOCATION\n");
154    printf("<A> to CLEAR MEMORY\t\t<P> to GET SCSI ERROR\n");
155    printf("<B> for CURS UPDATE DISABLE\t<T> to GET SAMPLING FREQUENCY\n");
156    printf("<C> for CURS VALUES\t\t<R> to RECORD\n");
157    printf("<D> for CURS UPDATE ENABLE\t<S> to STOP RECORD\n");
158    printf("<E> to TOGGLE TEXT ON/OFF\t<U> to SEND BUFFER....TO 5500\n");
159    printf("<F> to EXPRESS START\t\t<Y> to GET BUFFER.... FROM 5500\n");
160    printf("<G> to EXPRESS STOP\t\t<V> for INIT TRANS\n");
161    printf("<H> to RECALL SETUP TABLE\t<W> for WINDOW\n");
162    printf("<I> to GET 5500 VARBLOCK\t<X> for CHANGE IEEE ADDRESS\n");
163    printf("<J> to GET ANALYSIS FORMAT\t<Z> to GET TIME SLICE\n");
164    printf("<K> to GET SCREEN CHANNEL\t<Q> to quit.\n");
165    printf("<L> for LOCK 5500 KEYBOARD\n");
```

```
166     printf("<M> for MENU again\n");
167     printf("<N> to GET BUFFER SIZE\n");
168  }
169  TogglePrintSwitch()
170  {
171
172
173     /* when printEnableSet == 0, all diagnostic output of this
174        program is suppressed.  */
175
176     if (printEnableSet == 0)
177     {
178        printf("\nDiagnostic output enabled.\n");
179        printEnableSet = 1;
180     }
181
182     else
183     {
184        printEnableSet = 0;
185        printf("\nDiagnostic output disabled.\n");
186     }
187
188  }
189  sendCommand(parmsToSend)
190  int *parmsToSend;
191  {
192     int i,j,k;
193     char datum[20];
194
195     i = 0; j = 1;
196
197     i = *(parmsToSend + j);
198
199     if (printEnableSet == 1)
200     {
201        for (j = 0; j < i; j++)
202        {
203           printf("\nParameter %i: %i\n",j,*(parmsToSend + j));
204        }
205     }
206
207     /* output buffer to the translator */
208
209     ieeeprtf("output%.2d#%d buffer %d:%d EOI\n",transAddress,
210              (i * 2),segment(parmsToSend),offset(parmsToSend));
211
212
213     /* echo same thing to screen */
214
215     if (printEnableSet == 1)
216        printf("output%.2d#%d buffer %d:%d EOI\n",transAddress,
217              (i * 2),segment(parmsToSend),offset(parmsToSend));
218
219
220  }
```

```
221
222 UnlUntBus()
223 {
224
225     ieeewt("SEND UNT UNL\n");
226
227
228 }
229
230 GetSpollByte()
231 {
232
233     /* this routine loops until a SRQ is
234        detected, then reads the translator spoll byte. */
235
236     int spollByte = 0;
237
238
239
240     while (spollByte == 0)
241     {
242
243        ieeewt("spoll\n");
244
245        ieeescnf("%d", &spollByte);
246
247     }
248
249     ieeeprtf("spoll %.2d\n", transAddress);
250
251
252     ieeescnf("%d", &spollByte);
253
254     if (printEnableSet == 1)
255        printf("\nSpoll response: %d\n", spollByte);
256
257     return(spollByte);
258
259
260 }
261 WriteToTranslator(writeBufferStart,writeByteNumber)
262 int *writeBufferStart;
263 unsigned int *writeByteNumber;
264 {
265     /* Write from the designated data buffer the designated number
266        of bytes onto the IEEEbus. Assert "EOI" at the end of the
267        transmission */
268
269     /* write to translator */
270
271     ieeeprtf("output%.2d#%u buffer %d:%d EOI\n",transAddress, *writeByteNumber,
272        segment(writeBufferStart),offset(writeBufferStart));
273
274     /* echo to the screen */
275
```

```
276     if (printEnableSet == 1)
277        printf("output%.2d#%u buffer %d:%d EOI\n",transAddress, *writeByteNumber,
278           segment(writeBufferStart),offset(writeBufferStart));
279
280
281
282  }
283
284  ReadFromTranslator(readBufferStart,readByteNumber)
285  int *readBufferStart;
286  unsigned int *readByteNumber;
287  {
288     /* Read from the IEEEbus up to the designated number of
289        bytes and store them at the designated buffer location */
290
291     /* read from translator */
292
293     ieeeprtf("enter%.2d#%u buffer %d:%d EOI\n", transAddress, *readByteNumber,
294        segment(readBufferStart), offset(readBufferStart));
295
296     /* echo to the screen */
297
298     if (printEnableSet == 1)
299        printf("enter%.2d#%u buffer %d:%d EOI\n", transAddress, *readByteNumber,
300           segment(readBufferStart), offset(readBufferStart));
301
302
303
304
305  }
306  ReadIeeeBusWithBufferCount(readWCBuffer,readWCTotalBytes,readWCBytesReceived)
307  int *readWCBuffer;
308  unsigned int *readWCTotalBytes;
309  unsigned int *readWCBytesReceived;
310  {
311     /* Read from the IEEEbus up to the designated number of
312        bytes, store them at the designated buffer location,
313        report how many bytes were actually received. */
314
315
316     long int count;
317
318     /* read from translator */
319
320     ieeeprtf("enter%.2d#%u buffer %d:%d EOI\n", transAddress,
321        *readWCTotalBytes, segment(readWCBuffer), offset(readWCBuffer));
322
323     /* echo to the screen */
324
325     if (printEnableSet == 1)
326        printf("enter%.2d#%u buffer %d:%d EOI\n", transAddress,
327           *readWCTotalBytes, segment(readWCBuffer), offset(readWCBuffer));
328
329
330
```

```
331     ieeewt("buffered\n");
332     ieeescnf("%D",&count);
333
334     *readWCBytesReceived = (unsigned)count;
335
336     if (printEnableSet == 1)
337        printf("\nBytes received: %ld\n",count);
338
339  }
340
341  PutUpAWindow(PUAWBuffer)
342  int *PUAWBuffer;
343  {
344
345     char stringBuffer[63];
346     int buffer[20];
347     unsigned long int byteCount;
348     unsigned long int *bufferPointer;
349     unsigned int thct;
350     unsigned int *thisCount;
351     int errorCode;
352     char menuChoice, userchar;
353     int count;
354
355     struct WindowCommand
356     {
357        int identifier;
358        int parmCount;
359        unsigned int totalByteCount;
360     }
361
362     struct WindowCommand *wndPtr;
363
364     wndPtr = (struct WindowCommand*)buffer;
365
366     thisCount = &thct;
367
368     /* get a string */
369
370
371     printf("\nType in a string for display in a 5500 window.\n");
372
373     count = 0;
374
375     do
376     {
377        userchar = getche();
378        stringBuffer[count++] = userchar;
379
380
381     }
382     while ((count < 60)&&(userchar != '\r'));
383
384     count = count - 1;
385
```

```
386     stringBuffer[count] = '\0';
387
388     byteCount = count;
389
390     wndPtr->totalByteCount = (unsigned int)byteCount;
391
392     wndPtr->identifier = WINDOW_ON;  /* display window on 5500 screen */
393
394     wndPtr->parmCount = 3;           /* ints to send */
395
396     sendCommand(buffer);    /* send out the data */
397
398     UnlUntBus();
399
400     errorCode = GetSpollByte();
401
402
403     if (errorCode == 67)
404     {
405        /* translator wants us to send data */
406
407        *thisCount = (unsigned int)byteCount;
408
409        WriteToTranslator(stringBuffer,thisCount);
410
411        UnlUntBus();
412
413        errorCode = GetSpollByte();
414
415
416
417        if (errorCode == 64)
418        {
419
420           /* success */
421
422           printf("\nWINDOW_ON successful\n");
423
424           printf("\nPress any key to continue...\n");
425
426           getch();
427
428           wndPtr->identifier = WINDOW_OFF; /* turn off window on 5500 screen */
429
430           wndPtr->parmCount = 3;           /* ints to send */
431
432           sendCommand(buffer);    /* send out the data */
433
434           UnlUntBus();
435
436           errorCode = GetSpollByte();
437
438
439           if (errorCode == 67)
440           {
```

```
441             /* translator wants us to send data */
442
443             *thisCount = (unsigned int)byteCount;
444
445             WriteToTranslator(stringBuffer,thisCount);
446
447             UnlUntBus();
448
449             errorCode = GetSpollByte();
450
451
452             if(errorCode == 64)
453             {
454
455                /* success */
456
457                printf("\nWindow successfully removed\n");
458
459             }
460             else
461                printf("\nCould not remove window from 5500 screen.\n");
462
463          }
464          else
465          {
466             printf("\nSomething wrong with trans\n");
467          }
468
469
470       }
471       else
472          printf("\nSomething wrong with trans\n");
473
474
475    }
476    else
477       printf("\nSomething wrong with trans\n");
478
479
480
481
482
483 }
484 LockAndUnlockKeyboard(LAUKBuffer)
485 int *LAUKBuffer;
486 {
487
488
489    int buffer[20];
490    int errorCode;
491    int *byteCount;
492    int bc;
493
494    byteCount = &bc;
495
```

```
496     /* This routine will send the codes to lock and unlock
497        the 5500 keyboard */
498
499
500        buffer[0] = LOCK_5500_KBD; /* first location is call name */
501
502        buffer[1] = 2;          /* number of ints to send */
503
504        sendCommand(buffer); /* Send the string out on the bus */
505
506        UnlUntBus();
507
508        errorCode = GetSpollByte();
509
510        if (errorCode == 64)
511        {
512
513           printf("\nLOCK_5500_KBD successful.\n");
514
515           printf("\nPress any key to continue...\n");
516
517           getch();
518
519           buffer[0] = UNLOCK_5500_KBD;  /* first location is call name */
520
521           buffer[1] = 2;          /* number of ints to send */
522
523           sendCommand(buffer); /* Send the string out on the bus */
524
525           UnlUntBus();
526
527           errorCode = GetSpollByte();
528
529           if (errorCode == 64)
530           {
531
532              printf("\nUNLOCK_5500_KBD successful.\n");
533           }
534
535           else
536           {
537              printf("\nUNLOCK_5500_KBD not successful.\n");
538           }
539        }
540        else
541        {
542           printf("\nLOCK_5500_KBD not successful.\n");
543
544        }
545
546
547
548  }
549
550  Clear5500Memory(C5MBuffer)
```

```
551 int *C5MBuffer;
552 {
553    int buffer[20];
554    int errorCode;
555
556
557       buffer[0] = MEMORY_CLEAR;  /* first location is call name */
558
559       buffer[1] = 2;          /* number of ints to send */
560
561
562       sendCommand(buffer); /* Send the string out on the bus */
563
564       UnlUntBus();
565
566       errorCode = GetSpollByte();
567
568       if (errorCode == 64)
569       {
570
571          printf("\nMemory cleared successfully.\n");
572
573       }
574       else
575       {
576          printf("\nMemory NOT cleared.\n");
577
578       }
579
580
581 }
582
583
584 DisableCursorUpdates(DCUBuffer)
585 int *DCUBuffer;
586 {
587    int buffer[20];
588    int errorCode;
589
590
591       buffer[0] = CURSOR_UPDATE_DISABLE;  /* first location is call name */
592
593       buffer[1] = 2;          /* number of ints to send */
594
595
596       sendCommand(buffer); /* Send the string out on the bus */
597
598       UnlUntBus();
599
600       errorCode = GetSpollByte();
601
602       if (errorCode == 64)
603       {
604
605          printf("\nCursor updates disabled successfully.\n");
```

```
606         }
607         else
608         {
609            printf("\nCursor updates NOT disabled.\n");
610
611         }
612
613
614   }
615
616
617   EnableCursorUpdates(ECUBuffer)
618   int *ECUBuffer;
619   {
620      int buffer[20];
621      int errorCode;
622
623
624         buffer[0] = CURSOR_UPDATE_ENABLE;   /* first location is call name */
625
626         buffer[1] = 2;          /* number of ints to send */
627
628
629         sendCommand(buffer); /* Send the string out on the bus */
630
631         UnlUntBus();
632
633         errorCode = GetSpollByte();
634
635         if (errorCode == 64)
636         {
637
638            printf("\nCursor updates enabled successfully.\n");
639         }
640         else
641         {
642            printf("\nCursor updates NOT enabled.\n");
643
644         }
645
646
647   }
648
649   ExpressStart(EPBuffer)
650   int *EPBuffer;
651   {
652      int buffer[20];
653      int errorCode;
654      char userData;
655
656
657         printf("\nEnter <U> for UPPER, <L> for lower, <B> for BOTH screens.\n");
658
659         do
660         {
```

```
661            userData = getch();
662         }
663         while ((userData != 'U')&&(userData != 'L')&&(userData != 'B')&&
664             (userData != 'u')&&(userData != 'l')&&(userData != 'b'));
665
666
667         switch(userData)
668         {
669            case 'U':
670            case 'u':
671               buffer[2] = 1;
672               break;
673
674            case 'L':
675            case 'l':
676               buffer[2] = 0;
677               break;
678
679            case 'B':
680            case 'b':
681               buffer[2] = 2;
682               break;
683
684         }
685
686
687
688         buffer[0] = EXPRESS_START; /* first location is call name */
689
690         buffer[1] = 3;          /* number of ints to send */
691
692
693         sendCommand(buffer); /* Send the string out on the bus */
694
695         UnlUntBus();
696
697         errorCode = GetSpollByte();
698
699         if (errorCode == 64)
700         {
701
702            printf("\nExpress Start successful.\n");
703         }
704         else
705         {
706            printf("\nExpress Start Failed.\n");
707
708         }
709
710
711
712
713
714   }
715
```

```
716
717 ExpressStop(ESBuffer)
718 int *ESBuffer;
719 {
720    int buffer[20];
721    int errorCode;
722    char userData;
723
724    printf("\nEnter <U> for UPPER, <L> for lower, <B> for BOTH screens.\n");
725
726    do
727    {
728       userData = getch();
729    }
730    while ((userData != 'U')&&(userData != 'L')&&(userData != 'B')&&
731           (userData != 'u')&&(userData != 'l')&&(userData != 'b'));
732
733
734    switch(userData)
735    {
736       case 'U':
737       case 'u':
738          buffer[2] = 1;
739          break;
740
741       case 'L':
742       case 'l':
743          buffer[2] = 0;
744          break;
745
746       case 'B':
747       case 'b':
748          buffer[2] = 2;
749          break;
750
751    }
752
753
754    buffer[0] = EXPRESS_STOP;  /* first location is call name */
755
756    buffer[1] = 3;          /* number of ints to send */
757
758
759    sendCommand(buffer); /* Send the string out on the bus */
760
761    UnlUntBus();
762
763    errorCode = GetSpollByte();
764
765    if (errorCode == 64)
766    {
767
768       printf("\nExpress Stop successful.\n");
769    }
770    else
```

```
771     {
772        printf("\nExpress Stop Failed.\n");
773
774     }
775
776  }
777  RecallSetupTable(RSTBuffer)
778  int *RSTBuffer;
779  {
780     int buffer[20];
781     int errorCode;
782     unsigned int userData;
783
784     printf("\nEnter a setup number from 0 to 99.\n");
785
786     do
787     {
788        scanf("%d", &userData);
789     }
790     while ((userData < 0)||(userData > 99));
791
792     buffer[2] = userData;
793
794     buffer[0] = RECALL_SETUP_TABLE;  /* first location is call name */
795
796     buffer[1] = 3;          /* number of ints to send */
797
798
799     sendCommand(buffer); /* Send the string out on the bus */
800
801     UnlUntBus();
802
803     errorCode = GetSpollByte();
804
805     if (errorCode == 64)
806     {
807
808        printf("\nRecall setup successful.\n");
809     }
810     else
811     {
812        printf("\nRecall setup Failed.\n");
813
814     }
815
816  }
817  Get5500Varblock(G5VBuffer)
818  int *G5VBuffer;
819  {
820     int buffer[20];
821     int *commandString;
822     int errorCode;
823     int *dataSwapBuffer;
824     unsigned int nbytes;
825     unsigned int *byteCount;
```

```
826
827     byteCount = &nbytes;
828
829     dataSwapBuffer = G5VBuffer;
830
831     if (dataSwapBuffer == NULL)
832        printf("\nCalloc trouble");
833
834     buffer[0] = GET_5500_VARBLOCK;
835
836     buffer[1] = 2;          /* ints to send */
837
838     sendCommand(buffer);    /* send out the data */
839
840     UnlUntBus();
841
842     errorCode = GetSpollByte();
843
844     if (errorCode == 66)
845     {
846        /* translator wants us to have some data */
847
848        *byteCount = 0xFFF0;
849
850        ReadFromTranslator(dataSwapBuffer,byteCount);
851
852        UnlUntBus();
853
854        errorCode = GetSpollByte();
855
856        if (errorCode == 64)
857        {
858
859           /* success */
860
861           printf("\nStatus values received OK\n");
862
863           PrintSomeVarblockData(dataSwapBuffer);
864
865
866
867        }
868        else
869           printf("\nSomething wrong with trans\n");
870
871     }
872     else
873        printf("\nSomething wrong with trans\n");
874
875
876  }
877
878  PrintSomeVarblockData(varblockPointer)
879  int *varblockPointer;
880  {
```

```
881
882     struct keyvars_5500 *varPtr;
883
884
885     varPtr = (struct keyvars_5500*)varblockPointer;
886
887     /* print out some status */
888
889     if (varPtr->reconflag == 1)
890        printf("\nRecording right now.\n");
891
892     else
893        printf("\nNot recording right now.\n");
894
895
896     if (varPtr->timecursonflag == 1)
897        printf("\nTime cursors are on right now.\n");
898
899     else
900        printf("\nTime cursors are off right now.\n");
901
902
903  }
904  GetAnform(GABuffer)
905  int *GABuffer;
906  {
907     int buffer[20];
908     int *commandString;
909     int errorCode;
910     int *dataSwapBuffer;
911     unsigned int nbytes;
912     unsigned int *byteCount;
913     char userData;
914
915     byteCount = &nbytes;
916
917     dataSwapBuffer = GABuffer;
918
919     printf("\nChoose screen: <U> for upper, <L> for lower.\n");
920
921     do
922     {
923        userData = getch();
924     }
925     while ((userData != 'U')&&(userData != 'L')&&
926           (userData != 'u')&&(userData != 'l'));
927
928
929     switch(userData)
930     {
931        case 'U':
932        case 'u':
933           buffer[2] = UPPER_SCREEN;
934           break;
935
```

```
936         case 'L':
937         case 'l':
938            buffer[2] = LOWER_SCREEN;
939            break;
940
941
942      }
943
944
945      buffer[0] = GET_ANFORM;
946
947      buffer[1] = 3;          /* ints to send */
948
949      sendCommand(buffer);    /* send out the data */
950
951      UnlUntBus();
952
953      errorCode = GetSpollByte();
954
955      if (errorCode == 66)
956      {
957         /* translator wants us to have some data */
958
959         *byteCount = 0xFFF0;
960
961         ReadFromTranslator(dataSwapBuffer,byteCount);
962
963         UnlUntBus();
964
965         errorCode = GetSpollByte();
966
967         if (errorCode == 64)
968         {
969
970            /* success */
971
972            printf("\nScreen analysis format received OK\n");
973
974
975            switch(*(dataSwapBuffer + 0))
976            {
977
978               case NODISP_ANFORM:        /* No Display       */
979
980                  printf("\nNo Display on that screen.\n");
981                  break;
982
983
984               case PSPEC_ANFORM:         /* Power Spectrum   */
985                  printf("\nPower spectrum on that screen.\n");
986                  break;
987
988               case PWRBETC_ANFORM:      /* Power Between Cursors */
989                  printf("\nPower between cursors on that screen.\n");
990                  break;
```

```
  991
  992             case PWRATC_ANFORM:       /* Power At Cursors */
  993                printf("\nPower At cursors on that screen.\n");
  994                break;
  995
  996             case MINISON_ANFORM:      /* Minison Display  */
  997                printf("\nMinison display on that screen.\n");
  998                break;
  999
1,000             case WATER_ANFORM:        /* Waterfall        */
1,001                printf("\nWaterfall display on that screen.\n");
1,002                break;
1,003
1,004             case COMB_ANFORM:         /* Combination      */
1,005                printf("\nCombination display on that screen.\n");
1,006                break;
1,007
1,008             case AMP_ANFORM:          /* Amplitude        */
1,009                printf("\nAmplitude display on that screen.\n");
1,010                break;
1,011
1,012             case WAVE_ANFORM:         /* Waveform         */
1,013                printf("\nWaveform display on that screen.\n");
1,014                break;
1,015
1,016             case SONAGRAM_ANFORM:     /* Sonagram (TM)    */
1,017                printf("\nSonagram display on that screen.\n");
1,018                break;
1,019
1,020             case LTSA_ANFORM:         /* Long-Term Spectral Average */
1,021                printf("\nLong term spectral average on that screen.\n");
1,022                break;
1,023
1,024
1,025           } /* end of cases */
1,026
1,027       }
1,028       else
1,029          printf("\nSomething wrong with trans\n");
1,030
1,031    }
1,032    else
1,033       printf("\nSomething wrong with trans\n");
1,034
1,035
1,036
1,037 }
1,038
1,039 GetScreenChannel(GSCBuffer)
1,040 int *GSCBuffer;
1,041 {
1,042    int buffer[20];
1,043    int *commandString;
1,044    int errorCode;
1,045    int *dataSwapBuffer;
```

```
1,046     unsigned int nbytes;
1,047     unsigned int *byteCount;
1,048     char userData;
1,049
1,050     byteCount = &nbytes;
1,051
1,052
1,053     dataSwapBuffer = GSCBuffer;
1,054
1,055     printf("\nChoose screen: <U> for upper, <L> for lower.\n");
1,056
1,057     do
1,058     {
1,059        userData = getch();
1,060     }
1,061     while ((userData != 'U')&&(userData != 'L')&&
1,062            (userData != 'u')&&(userData != 'l'));
1,063
1,064
1,065     switch(userData)
1,066     {
1,067        case 'U':
1,068        case 'u':
1,069           buffer[2] = UPPER_SCREEN;
1,070           break;
1,071
1,072        case 'L':
1,073        case 'l':
1,074           buffer[2] = LOWER_SCREEN;
1,075           break;
1,076
1,077
1,078     }
1,079
1,080
1,081     buffer[0] = GET_SCREEN_CHANNEL;
1,082
1,083     buffer[1] = 3;          /* ints to send */
1,084
1,085     sendCommand(buffer);    /* send out the data */
1,086
1,087     UnlUntBus();
1,088
1,089     errorCode = GetSpollByte();
1,090
1,091     if (errorCode == 66)
1,092     {
1,093        /* translator wants us to have some data */
1,094
1,095        *byteCount = 0xFFF0;
1,096
1,097        ReadFromTranslator(dataSwapBuffer,byteCount);
1,098
1,099        UnlUntBus();
1,100
```

```
1,101        errorCode = GetSpollByte();
1,102
1,103        if (errorCode == 64)
1,104        {
1,105
1,106           /* success */
1,107
1,108           printf("\nScreen channel received OK\n");
1,109
1,110           if (*(dataSwapBuffer + 0) == 1)
1,111              printf("\nChannel 1 is on the screen you chose.\n");
1,112
1,113           else
1,114              printf("\nChannel 2 is on the screen you chose.\n");
1,115
1,116
1,117        }
1,118        else
1,119           printf("\nSomething wrong with trans\n");
1,120
1,121     }
1,122     else
1,123        printf("\nSomething wrong with trans\n");
1,124
1,125
1,126
1,127  }
1,128
1,129  GetBufferSize(GBSBuffer)
1,130  int *GBSBuffer;
1,131  {
1,132     int buffer[20];
1,133     int *commandString;
1,134     int errorCode;
1,135     int *dataSwapBuffer;
1,136     unsigned int nbytes;
1,137     unsigned int *byteCount;
1,138     int userData;
1,139
1,140     byteCount = &nbytes;
1,141
1,142
1,143     GBSBuffer = dataSwapBuffer;
1,144
1,145     printf("\nChoose channel: <1> for Channel 1, <2> for Channel 2.\n");
1,146
1,147     do
1,148     {
1,149        scanf("%d", &userData);
1,150     }
1,151     while ((userData != 1)&&(userData != 2));
1,152
1,153
1,154     buffer[2] = userData;
1,155
```

```
1,266  GetScsiError(GSEBuffer)
1,267  int *GSEBuffer;
1,268  {
1,269     int buffer[20];
1,270     int *commandString;
1,271     int errorCode;
1,272     int *dataSwapBuffer;
1,273     unsigned int nbytes;
1,274     unsigned int *byteCount;
1,275     int userData;
1,276
1,277     byteCount = &nbytes;
1,278
1,279     dataSwapBuffer = GSEBuffer;
1,280
1,281     buffer[0] = SCSI_ERROR;
1,282
1,283     buffer[1] = 2;          /* ints to send */
1,284
1,285     sendCommand(buffer);    /* send out the data */
1,286
1,287     UnlUntBus();
1,288
1,289     errorCode = GetSpollByte();
1,290
1,291     if (errorCode == 66)
1,292     {
1,293        /* translator wants us to have some data */
1,294
1,295        *byteCount = 0xFFF0;
1,296
1,297        ReadFromTranslator(dataSwapBuffer,byteCount);
1,298
1,299        UnlUntBus();
1,300
1,301        errorCode = GetSpollByte();
1,302
1,303        if (errorCode == 64)
1,304        {
1,305
1,306           /* success */
1,307
1,308           printf("\nMessage received OK\n");
1,309
1,310           printf("\nSCSI error message: %s \n", (char*)dataSwapBuffer);
1,311
1,312        }
1,313        else
1,314           printf("\nSomething wrong with trans\n");
1,315
1,316     }
1,317     else
1,318        printf("\nSomething wrong with trans\n");
1,319
1,320
```

```
1,321  }
1,322
1,323  GetSamplingFrequency(GSFBuffer)
1,324  int *GSFBuffer;
1,325  {
1,326     int buffer[20];
1,327     int *commandString;
1,328     int errorCode;
1,329     int *dataSwapBuffer;
1,330     int nbytes;
1,331     int *byteCount;
1,332     int userData;
1,333     unsigned int *byteAlignCount;
1,334     unsigned int bac;
1,335
1,336     byteAlignCount = &bac;
1,337
1,338     byteCount = &nbytes;
1,339
1,340     dataSwapBuffer = GSFBuffer;
1,341
1,342     printf("\nChoose channel: <1> for Channel 1, <2> for Channel 2.\n");
1,343
1,344     do
1,345     {
1,346        scanf("%d", &userData);
1,347     }
1,348     while ((userData != 1)&&(userData != 2));
1,349
1,350     buffer[2] = userData;
1,351
1,352     buffer[0] = GET_SAMP_FREQ;
1,353
1,354     buffer[1] = 3;          /* ints to send */
1,355
1,356     sendCommand(buffer);    /* send out the data */
1,357
1,358     UnlUntBus();
1,359
1,360     errorCode = GetSpollByte();
1,361
1,362     if (errorCode == 66)
1,363     {
1,364        /* translator wants us to have some data */
1,365
1,366        *byteCount = 0xFFF0;
1,367
1,368        ReadFromTranslator(dataSwapBuffer,byteCount);
1,369
1,370        UnlUntBus();
1,371
1,372        errorCode = GetSpollByte();
1,373
1,374        if (errorCode == 64)
1,375        {
```

```
1,376
1,377          /* success */
1,378
1,379          printf("\nSampling frequency received OK\n");
1,380
1,381          printf("\nThe sampling frequency for channel %d is: %ld\n",
1,382                userData, *(long int*)dataSwapBuffer);
1,383
1,384       }
1,385       else
1,386          printf("\nSomething wrong with trans\n");
1,387
1,388    }
1,389    else
1,390       printf("\nSomething wrong with trans\n");
1,391
1,392
1,393
1,394
1,395
1,396 }
1,397 InitTrans(ITBuffer)
1,398 int *ITBuffer;
1,399 {
1,400    int buffer[20];
1,401    int errorCode;
1,402    int bc;
1,403    int *byteCount;
1,404
1,405    byteCount = &bc;
1,406
1,407
1,408    buffer[0] = INIT_TRANS; /* first location is call name */
1,409
1,410    buffer[1] = 2;          /* number of ints to send */
1,411
1,412    sendCommand(buffer); /* Send the string out on the bus */
1,413
1,414
1,415    UnlUntBus();
1,416
1,417    errorCode = GetSpollByte();
1,418
1,419    if (errorCode == 64)
1,420    {
1,421
1,422       printf("\nInit Trans successful.\n");
1,423    }
1,424    else
1,425    {
1,426       printf("\nInit Trans Failed.\n");
1,427
1,428    }
1,429
1,430
```

```
1,431
1,432
1,433
1,434  }
1,435
1,436  ByteAlign(baBuffer, baCount)
1,437  int *baBuffer;
1,438  unsigned int *baCount;
1,439  {
1,440
1,441     unsigned int count;
1,442     char temp;
1,443     int *buffer;
1,444
1,445     struct align
1,446     {
1,447        char byte1;
1,448        char byte2;
1,449     };
1,450
1,451     struct align *alignPtr;
1,452
1,453     buffer = baBuffer;
1,454
1,455     for (count = 0; count < *baCount; count++)
1,456     {
1,457        buffer = baBuffer + count;
1,458
1,459        alignPtr = (struct align*)buffer;
1,460
1,461        temp = alignPtr->byte1;
1,462        alignPtr->byte1 = alignPtr->byte2;
1,463        alignPtr->byte2 = temp;
1,464
1,465     }
1,466
1,467  }
1,468  IntAlign(iaBuffer, iaCount)
1,469  long int *iaBuffer;
1,470  unsigned int *iaCount;
1,471  {
1,472
1,473     unsigned int count;
1,474     int temp;
1,475     long int *buffer;
1,476
1,477     struct alignInt
1,478     {
1,479        int intVal1;
1,480        int intVal2;
1,481     };
1,482
1,483     struct alignInt *alignIntPtr;
1,484
1,485     buffer = iaBuffer;
```

```
1,486
1,487     for (count = 0; count < *iaCount; count++)
1,488     {
1,489        buffer = iaBuffer + count;
1,490
1,491        alignIntPtr = (struct alignInt*)buffer;
1,492
1,493        temp = alignIntPtr->intVal1;
1,494        alignIntPtr->intVal1 = alignIntPtr->intVal2;
1,495        alignIntPtr->intVal2 = temp;
1,496
1,497     }
1,498
1,499  }
1,500  ChangeIeeeAddress(CIABuffer)
1,501  int *CIABuffer;
1,502  {
1,503     int buffer[20];
1,504     int errorCode;
1,505     int userData;
1,506
1,507     /* input new address */
1,508
1,509     printf("\nEnter a number from 00 to 30 for new Translator address.\n");
1,510
1,511     do
1,512     {
1,513        scanf("%d", &userData);
1,514     }
1,515     while ((userData < 0)||(userData > 30));
1,516
1,517
1,518     /* send new address to trans */
1,519
1,520
1,521
1,522        buffer[0] = CHANGE_TRANS_ADDRESS;   /* first location is call name */
1,523
1,524        buffer[1] = 3;          /* number of ints to send */
1,525
1,526        buffer[2] = userData;
1,527
1,528
1,529        sendCommand(buffer); /* Send the string out on the bus */
1,530
1,531        UnlUntBus();
1,532
1,533        transAddress = userData;
1,534
1,535
1,536        errorCode = GetSpollByte();
1,537
1,538        if (errorCode == 64)
1,539        {
1,540
```

```
1,541          printf("\nNew address accepted.\n");
1,542
1,543       }
1,544       else
1,545       {
1,546          printf("\nAddress not changed.\n");
1,547
1,548       }
1,549
1,550
1,551
1,552
1,553 }
1,554
1,555 GetBuffer(GBBuffer)
1,556 int *GBBuffer;
1,557 {
1,558    int buffer[20];
1,559    int *commandString;
1,560    int errorCode;
1,561    int *dataSwapBuffer;
1,562    unsigned int nbytes;
1,563    unsigned int *byteCount;
1,564    char userData;
1,565    FILE *stream;
1,566    int handle, count1, count2, userNum;
1,567
1,568    byteCount = &nbytes;
1,569
1,570    dataSwapBuffer = GBBuffer;
1,571
1,572    /* open a file for spooling data */
1,573
1,574    if ((stream = fopen("5500.dta", "w+b")) == NULL)
1,575    {
1,576       printf("\nCould not open file.\n");
1,577       return(0);
1,578    }
1,579
1,580    printf("\nEnter the source channel. <1> Channel 1, <2> Channel 2.\n");
1,581
1,582    do
1,583    {
1,584       scanf("%d", &userNum);
1,585    }
1,586    while ((userNum != 1)&&(userNum != 2));
1,587
1,588    if (userNum == 1)
1,589       buffer[2] =  CHANNEL_1;
1,590
1,591    if (userNum == 2)
1,592       buffer[2] =  CHANNEL_2;
1,593
1,594
1,595
```

```
1,596     printf("\nEnter bit resolution. <8>, <10>, <12>, <16>.\n");
1,597
1,598     do
1,599     {
1,600        scanf("%d", &userNum);
1,601     }
1,602     while ((userNum != 8)&&(userNum != 10)&&(userNum != 12)&&(userNum != 16));
1,603
1,604
1,605     buffer[3] = userNum;
1,606
1,607     buffer[0] = GET_BUFFER;
1,608
1,609     buffer[1] = 5;          /* ints to send */
1,610
1,611     buffer[4] = FALSE;      /* neg flag is zed */
1,612
1,613     sendCommand(buffer);    /* send out the data */
1,614
1,615     UnlUntBus();
1,616
1,617     errorCode = GetSpollByte();
1,618
1,619     if (errorCode == 66)
1,620     {
1,621        do
1,622        {
1,623           /* translator wants us to have some data */
1,624
1,625           *byteCount = 0xFFF0;
1,626
1,627           printf("Received Data\n");
1,628
1,629           ReadIeeeBusWithBufferCount(dataSwapBuffer,byteCount,byteCount);
1,630
1,631           UnlUntBus();
1,632
1,633           /* write *byteCount of data to disk */
1,634
1,635
1,636           if (buffer[3] > 8)
1,637           {
1,638              count1 = *byteCount / 2;
1,639              printf("\n Count1 = %d" ,count1);
1,640              errorCode = fwrite(dataSwapBuffer, sizeof(int), count1, stream);
1,641           }
1,642
1,643           else
1,644           {
1,645              count1 = *(int*)byteCount;
1,646              printf("\n Count1 = %d" ,count1);
1,647              errorCode = fwrite(dataSwapBuffer, sizeof(char), count1, stream);
1,648           }
1,649
1,650
```

```
1,651          errorCode = GetSpollByte();
1,652       }
1,653       while (errorCode == 66);
1,654
1,655       if (errorCode == 64)
1,656       {
1,657          /* success */
1,658
1,659          printf("\nData transferred successfully.\n");
1,660
1,661       }
1,662       else
1,663       {
1,664          /* error */
1,665
1,666          printf("\nError occurred.\n");
1,667       }
1,668
1,669    }
1,670    else
1,671       printf("\nError occurred.\n");
1,672
1,673
1,674    /* close file */
1,675
1,676    fclose(stream);
1,677
1,678
1,679 }
1,680 SendBuffer(SBBuffer)
1,681 int *SBBuffer;
1,682 {
1,683
1,684    int buffer[20];
1,685    int *commandString;
1,686    int errorCode;
1,687    int *dataSwapBuffer;
1,688    unsigned int nbytes;
1,689    unsigned int *byteCount;
1,690    char userData;
1,691    FILE *stream;
1,692    int *count;
1,693    int ct, userNum;
1,694    long unsigned *intbuff;
1,695
1,696
1,697    struct SendBufferData
1,698    {
1,699       int identifier;
1,700       int parmCount;
1,701       long unsigned length;
1,702       int destination;
1,703       int numbits;
1,704       int neg_flag;
1,705    };
```

```
1,706
1,707     struct SendBufferData *sbPtr;
1,708
1,709     byteCount = &nbytes;
1,710
1,711     sbPtr = (struct SendBufferData*)buffer;
1,712
1,713     count = &ct;
1,714
1,715     dataSwapBuffer = SBBuffer;
1,716
1,717
1,718     /* open a file for spooling data */
1,719
1,720     if ((stream = fopen("5500.dta", "rb")) == NULL)
1,721     {
1,722        printf("\nCould not open file.\n");
1,723        return(0);
1,724     }
1,725
1,726     sbPtr->length = (long unsigned)(filelength(fileno(stream)));
1,727
1,728
1,729     sbPtr->identifier =  SEND_BUFFER;
1,730
1,731     sbPtr->parmCount = 7;            /* ints to send */
1,732
1,733     printf("\nEnter the destination channel. <1> Channel 1, <2> Channel 2.\n");
1,734
1,735     do
1,736     {
1,737        scanf("%d", &userNum);
1,738     }
1,739     while ((userNum != 1)&&(userNum != 2));
1,740
1,741     if (userNum == 1)
1,742        sbPtr->destination =  CHANNEL_1;
1,743
1,744     if (userNum == 2)
1,745        sbPtr->destination =  CHANNEL_2;
1,746
1,747
1,748
1,749     printf("\nEnter bit resolution. <8>, <10>, <12>, <16>.\n");
1,750
1,751     do
1,752     {
1,753        scanf("%d", &userNum);
1,754     }
1,755     while ((userNum != 8)&&(userNum != 10)&&(userNum != 12)&&(userNum != 16));
1,756
1,757
1,758     sbPtr->numbits = userNum;
1,759
1,760     sbPtr->neg_flag = FALSE;      /* neg flag is zed */
```

```
1,761
1,762     sendCommand(buffer);    /* send out the data */
1,763
1,764     UnlUntBus();
1,765
1,766     errorCode = GetSpollByte();
1,767
1,768
1,769     if (errorCode == 67)
1,770     {
1,771        do
1,772        {
1,773           if (sbPtr->numbits > 8)
1,774           {
1,775              errorCode = fread(dataSwapBuffer, sizeof(int), 0x3FFF, stream);
1,776              errorCode = errorCode * 2;
1,777           }
1,778           else
1,779           {
1,780              errorCode = fread(dataSwapBuffer, sizeof(char), 0x7FFF, stream);
1,781
1,782           }
1,783
1,784           printf("\nSending %d bytes\n", errorCode);
1,785
1,786           *byteCount = 0;
1,787
1,788           *(int*)byteCount = errorCode;
1,789
1,790           WriteToTranslator(dataSwapBuffer,byteCount);
1,791
1,792           UnlUntBus();
1,793
1,794           printf("Sent data\n");
1,795
1,796           errorCode = GetSpollByte();
1,797        }
1,798        while (errorCode == 67);
1,799
1,800        if (errorCode == 64)
1,801        {
1,802           /* success */
1,803
1,804           printf("\nData transferred successfully.\n");
1,805
1,806        }
1,807        else
1,808        {
1,809           /* error */
1,810
1,811           printf("\nError occurred.\n");
1,812        }
1,813
1,814     }
1,815     else
```

```
1,816       printf("\nError occurred.\n");
1,817
1,818    fclose(stream);
1,819
1,820
1,821 }
1,822 GetTimeSlice(GTSBuffer)
1,823 int *GTSBuffer;
1,824 {
1,825    int buffer[20];
1,826    int *commandString;
1,827    int errorCode;
1,828    int *dataSwapBuffer;
1,829    unsigned int nbytes;
1,830    unsigned int *byteCount;
1,831    int userData;
1,832    FILE *stream;
1,833    int count, returnValue;
1,834    int *comCount;
1,835    int cc;
1,836
1,837    comCount = &cc;
1,838    byteCount = &nbytes;
1,839
1,840    dataSwapBuffer = GTSBuffer;
1,841
1,842    printf("SOURCE: <1> for upper screen, <2> for lower screen, <3> for red cursor.\n");
1,843
1,844
1,845    do
1,846    {
1,847       scanf("%d", &userData);
1,848    }
1,849    while((userData < 1)||(userData > 3));
1,850
1,851    switch(userData)
1,852    {
1,853       case 1:  buffer[2] = UPPER_SCREEN; break;
1,854
1,855       case 2:  buffer[2] = LOWER_SCREEN; break;
1,856
1,857       case 3:  buffer[2] = RED_CURSOR; break;
1,858    }
1,859
1,860    printf("\nDo you want the KAY format header? (<1> yes, <0> no.)\n");
1,861
1,862    do
1,863    {
1,864       scanf("%d", &userData);
1,865    }
1,866    while((userData < 0)||(userData > 1));
1,867
1,868    buffer[3] = userData;
1,869
1,870    buffer[0] = GET_TIME_SLICE;
```

```
1,871
1,872     buffer[1] = 4;          /* ints to send */
1,873
1,874     sendCommand(buffer);    /* send out the data */
1,875
1,876     UnlUntBus();
1,877
1,878     errorCode = GetSpollByte();
1,879
1,880     if (errorCode == 66)
1,881     {
1,882        /* translator wants us to have some data */
1,883
1,884        *byteCount = 0xFFF0;
1,885
1,886        ReadIeeeBusWithBufferCount(dataSwapBuffer,byteCount,byteCount);
1,887
1,888        UnlUntBus();
1,889
1,890        printf("Trans has sent us %u bytes.\n", *byteCount);
1,891
1,892        errorCode = GetSpollByte();
1,893
1,894        if (errorCode == 64)
1,895        {
1,896
1,897           /* success */
1,898
1,899           printf("\nSpectral data received OK\n");
1,900
1,901           printf("Saving it to disk.\n");
1,902
1,903           stream = fopen("spectest.dta", "w+b");
1,904
1,905           count = *(int *)byteCount;
1,906
1,907           returnValue = fwrite((char *)dataSwapBuffer, sizeof(char),
1,908                    count, stream);
1,909
1,910           if (returnValue == count)
1,911              printf("\nFile saved OK.\n");
1,912
1,913           fclose(stream);
1,914
1,915
1,916        }
1,917        else
1,918           printf("\nSomething wrong with trans\n");
1,919
1,920     }
1,921     else
1,922        printf("\nSomething wrong with trans\n");
1,923
1,924
1,925  }
```

```
1,926  PressRecordButton(PRBBuffer)
1,927  int *PRBBuffer;
1,928  {
1,929     int *dataSwapBuffer;
1,930     int buffer[20];
1,931     int errorCode;
1,932
1,933
1,934        /* "RECORD_ON" has been selected. Send that option
1,935           out onto the IEEE bus */
1,936
1,937        buffer[0] = KEY_DOWN;   /* first location is call name */
1,938
1,939        buffer[1] = 6;          /* number of ints to send */
1,940
1,941        buffer[2] = RECORD_ON;  /* Button to press */
1,942
1,943        buffer[3] = 1;          /* Press one time */
1,944
1,945        buffer[4] = RECORD_ON;  /* repeat of original key means
1,946                                   that is the only key to press.
1,947                                   A different key means press both
1,948                                   simultaneously. */
1,949
1,950
1,951        buffer[5] = 0;       /* currently unused */
1,952
1,953
1,954        sendCommand(buffer); /* Send the string out on the bus */
1,955
1,956        UnlUntBus();
1,957
1,958        errorCode = GetSpollByte();
1,959
1,960        if (errorCode == 64)
1,961        {
1,962
1,963           printf("\nKEY_DOWN successful.\n");
1,964
1,965           buffer[0] = ALL_KEYS_UP;   /* first location is call name */
1,966
1,967           buffer[1] = 2;          /* number of ints to send */
1,968
1,969
1,970
1,971           sendCommand(buffer); /* Send the string out on the bus */
1,972
1,973           UnlUntBus();
1,974
1,975           errorCode = GetSpollByte();
1,976
1,977           if (errorCode == 64)
1,978           {
1,979              printf("\nALL_KEYS_UP successful.\n");
1,980
```

```
1,981          }
1,982          else
1,983          {
1,984             printf("\nALL_KEYS_UP failed.\n");
1,985          }
1,986       }
1,987       else
1,988       {
1,989          printf("\nKEY_DOWN failed.\n");
1,990       }
1,991
1,992
1,993
1,994
1,995
1,996 }
1,997 PressStopButton(PSBBuffer)
1,998 int *PSBBuffer;
1,999 {
2,000    int *dataSwapBuffer;
2,001    int buffer[20];
2,002    int errorCode;
2,003
2,004
2,005       /* "RECORD_OFF" has been selected. Send that option
2,006          out onto the IEEE bus */
2,007
2,008
2,009       buffer[0] = KEY_DOWN;   /* first location is call name */
2,010
2,011       buffer[1] = 6;          /* number of ints to send */
2,012
2,013       buffer[2] = RECORD_OFF; /* Button to press */
2,014
2,015       buffer[3] = 1;          /* Press one time */
2,016
2,017       buffer[4] = RECORD_OFF; /* repeat of original key means
2,018                                  that is the only key to press.
2,019                                  A different key means press both
2,020                                  simultaneously. */
2,021
2,022
2,023       buffer[5] = 0;          /* currently unused */
2,024
2,025
2,026       sendCommand(buffer); /* Send the string out on the bus */
2,027
2,028       UnlUntBus();
2,029
2,030
2,031       errorCode = GetSpollByte();
2,032
2,033       if (errorCode == 64)
2,034       {
2,035          printf("\nKEY_DOWN successful.\n");
```

```
2,036
2,037          buffer[0] = ALL_KEYS_UP;   /* first location is call name */
2,038
2,039          buffer[1] = 2;          /* number of ints to send */
2,040
2,041          sendCommand(buffer); /* Send the string out on the bus */
2,042
2,043          UnlUntBus();
2,044
2,045          errorCode = GetSpollByte();
2,046
2,047          if (errorCode == 64)
2,048          {
2,049             printf("\nALL_KEYS_UP successful.\n");
2,050
2,051          }
2,052          else
2,053          {
2,054             printf("\nALL_KEYS_UP failed.\n");
2,055          }
2,056       }
2,057       else
2,058       {
2,059          printf("\nKEY_DOWN failed.\n");
2,060       }
2,061
2,062
2,063 }
2,064 TransExit(TEBuffer)
2,065 int *TEBuffer;
2,066 {
2,067    int *dataSwapBuffer;
2,068    int buffer[20];
2,069    int errorCode;
2,070
2,071    /* send a code that makes the translator bail out */
2,072
2,073    buffer[0] = TRANS_EXIT; /* bail out code */
2,074
2,075    buffer[1] = 2;          /* ints to send */
2,076
2,077    sendCommand(buffer);    /* send out the data */
2,078
2,079    UnlUntBus();
2,080
2,081    errorCode = GetSpollByte();
2,082
2,083    if (errorCode == 70)
2,084    {
2,085       printf("\nTRANS_EXIT successful.\n");
2,086
2,087    }
2,088    else
2,089    {
2,090       printf("\nTRANS_EXIT failed.\n");
```

```
2,091     }
2,092
2,093
2,094  }
2,095
2,096  GetCursorValues(GCVBuffer)
2,097  int *GCVBuffer;
2,098  {
2,099     int *dataSwapBuffer;
2,100     int buffer[20];
2,101     int errorCode;
2,102     unsigned int nbytes;
2,103     unsigned int *byteCount;
2,104
2,105     dataSwapBuffer = GCVBuffer;
2,106
2,107     byteCount = &nbytes;
2,108
2,109     /* send a code that gets a CURSOR_VALUES block */
2,110
2,111     buffer[0] = CURS_VALS;  /* get cursor vals */
2,112
2,113     buffer[1] = 2;          /* ints to send */
2,114
2,115     sendCommand(buffer);    /* send out the data */
2,116
2,117     UnlUntBus();
2,118
2,119     errorCode = GetSpollByte();
2,120
2,121     if (errorCode == 66)
2,122     {
2,123        /* translator wants us to have some data */
2,124
2,125        *byteCount = 0xFFF0;
2,126
2,127        ReadFromTranslator(dataSwapBuffer,byteCount);
2,128
2,129        UnlUntBus();
2,130
2,131        errorCode = GetSpollByte();
2,132
2,133        if (errorCode == 64)
2,134        {
2,135
2,136           /* success */
2,137
2,138           printf("\nCursor values received OK\n");
2,139
2,140           PrintOutSomeCursorInfo(dataSwapBuffer);
2,141
2,142
2,143
2,144        }
2,145        else
```

```
2,146           printf("\nSomething wrong with trans\n");
2,147
2,148       }
2,149       else
2,150          printf("\nSomething wrong with trans\n");
2,151
2,152
2,153   }
2,154   PrintOutSomeCursorInfo(cursorBufferStart)
2,155   char *cursorBufferStart;
2,156   {
2,157
2,158       char deltat[20];
2,159
2,160       /* set up pointer to structure */
2,161
2,162
2,163       struct CURSOR_VALUES *curs_ptr;
2,164
2,165       curs_ptr = (struct CURSOR_VALUES*)cursorBufferStart;
2,166
2,167       /*  print out delta t */
2,168
2,169       strncpy(deltat,curs_ptr->deltat,17);
2,170       deltat[17] = '\0';
2,171
2,172       printf("\nTime between cursors: %s\n",deltat);
2,173
2,174   }
2,175   PressAnyKey(PAKBuffer)
2,176   int *PAKBuffer;
2,177   {
2,178       int userChoice;
2,179       int *dataSwapBuffer;
2,180       int buffer[20];
2,181       int errorCode;
2,182
2,183       userChoice = GetTheKey();
2,184
2,185       /* press the chosen key */
2,186
2,187          buffer[0] = KEY_DOWN;   /* first location is call name */
2,188
2,189          buffer[1] = 6;          /* number of ints to send */
2,190
2,191          buffer[2] = userChoice; /* Button to press */
2,192
2,193          buffer[3] = 1;          /* Press one time */
2,194
2,195          buffer[4] = userChoice; /* repeat of original key means
2,196                                     that is the only key to press.
2,197                                     A different key means press both
2,198                                     simultaneously. */
2,199
2,200
```

```
2,201        buffer[5] = 0;          /* currently unused */
2,202
2,203
2,204        sendCommand(buffer); /* Send the string out on the bus */
2,205
2,206        UnlUntBus();
2,207
2,208
2,209        errorCode = GetSpollByte();
2,210
2,211        if (errorCode == 64)
2,212        {
2,213           printf("\nKEY_DOWN successful.\n");
2,214
2,215           buffer[0] = ALL_KEYS_UP;   /* first location is call name */
2,216
2,217           buffer[1] = 2;          /* number of ints to send */
2,218
2,219           sendCommand(buffer); /* Send the string out on the bus */
2,220
2,221           UnlUntBus();
2,222
2,223           errorCode = GetSpollByte();
2,224
2,225           if (errorCode == 64)
2,226           {
2,227              printf("\nALL_KEYS_UP successful.\n");
2,228
2,229           }
2,230           else
2,231           {
2,232              printf("\nALL_KEYS_UP failed.\n");
2,233           }
2,234        }
2,235        else
2,236        {
2,237           printf("\nKEY_DOWN failed.\n");
2,238        }
2,239
2,240
2,241
2,242  }
2,243  GetTheKey()
2,244  {
2,245
2,246     int userData;
2,247
2,248
2,249     printf("CH1:10dB down...00\t\CH1:1dB down...01\tCH1:coupling...02\n");
2,250     printf("CH2:coupling...03\tCH2:10dB down...04\tCH2:1dB down...05\n");
2,251     printf("CH2:10dB up...06\tCH2:1dB up...07\t\tCH1:volume down...08\n");
2,252     printf("CH1:volume up...09\tCH2:volume up...10\tRECORD OFF...11\n");
2,253     printf("STOP...12\t\tCH2:volume down...13\tRECORD ON...14\n");
2,254     printf("Time Cursor Left...16\tTime Cursor Right...17\tFreq. Cursor Store...18\n");
2,255     printf("Freq Cursor up...19\tFreq Cursor Clear...20\tFreq Cursor down...21\n");
```

```
2,256     printf("Scroll up left...22\tScroll Low Left...23\tPrint...25\n");
2,257     printf("Function A...26\t\tFunction B...27\t\tFunction C...28\n");
2,258     printf("Function D...29\t\tFunction E...30\t\tFunction F...31\n");
2,259     printf("Freq Cursor on/off...32\tCH1:1dB up...34\t\tCH1:10dB up...35\n");
2,260     printf("Time Cursor up/low...36\tTime Cursor on/off...37\tTime Cursor Store...38\n");
2,261     printf("Time Cursor Clear...39\tMenu Select...40\tCH2:Mute...41\n");
2,262     printf("Menu position up...42\tMenu on/off...43\tButton 9...44\n");
2,263     printf("Button 7...45\t\tMenu Position L/R...46\tMenu Position down...47\n");
2,264     printf("Decimal Point...48\tButton 1...49\t\tMono Stereo...50\n");
2,265     printf("Ch1 Mute...51\t\tScroll both right...52\tScroll Both Left...53\n");
2,266     printf("Scroll up Right...54\tScroll low right...55\tEnter...56\n");
2,267     printf("Button 0...57\t\tButton 2...58\t\tButton 3...59\n");
2,268     printf("Button 4...60\t\tButton 5...61\t\tButton 6...62\n");
2,269     printf("Button 8...63\n");
2,270
2,271     do
2,272     {
2,273        scanf("%d", &userData);
2,274     }
2,275     while ((userData < 0)||(userData > 63)||(userData == 15)||(userData == 24)
2,276           ||(userData == 33));
2,277
2,278
2,279     return(userData);
2,280
2,281
2,282  }
```

```
#include <dos.h>
#include <stdio.h>
#include <malloc.h>
#include <bios.h>
#include        <sys\types.h>
#include        <sys\stat.h>
#include <io.h>
#include "ieeeio.h"
#include        "keys.h"
#include "scsiuser.h"
#include "key_5500.h"
#include "kay_fmt.h"



// here are definitions for various function calls

#define TRANS_EXIT 0
#define KEY_DOWN 1
#define ALL_KEYS_UP 2
#define LOCK_5500_KBD 3
#define UNLOCK_5500_KBD 4
#define MEMORY_CLEAR 5
#define CURSOR_UPDATE_DISABLE 6
#define CURSOR_UPDATE_ENABLE 7
#define EXPRESS_START 8
#define EXPRESS_STOP 9
#define RECALL_SETUP_TABLE 10
#define WINDOW_ON 11
#define WINDOW_OFF 12
#define GET_5500_VARBLOCK 13
#define GET_ANFORM 14
#define GET_SCREEN_CHANNEL 15
#define GET_BUFFER_SIZE 16
#define CURS_VALS 17
#define GET_TIME_SLICE 18
#define GET_VCBFLAG 19
#define GET_BUFFER 20
#define SEND_BUFFER 21
#define SCSI_ERROR 22
#define GET_SAMP_FREQ 23
#define CHANGE_TRANS_ADDRESS 24
#define INIT_TRANS 25



// key press defines

#define RECORD_ON 14
#define RECORD_OFF 11
```